Rethinking System Design for Expressive Cryptography

By

Sam Kumar

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor David E. Culler, Co-chair
Associate Professor Raluca Ada Popa, Co-chair
Professor Scott Shenker
Professor Chris Jay Hoofnagle

Summer 2023

Rethinking System Design for Expressive Cryptography

Copyright 2023
by
Sam Kumar

Abstract

Rethinking System Design for Expressive Cryptography

by

Sam Kumar

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor David E. Culler, Co-chair

Associate Professor Raluca Ada Popa, Co-chair

Expressive cryptography, including Secure Multi-Party Computation (SMPC), Fully Homomorphic Encryption (FHE), and policy-based encryption, has the potential to enable transformative new applications. Unfortunately, it is often slow and resource-intensive, making those applications difficult to realize. For example, SMPC enables multiple organizations (e.g., hospitals) to run joint computations on their data (e.g., for better medical diagnosis and treatment) while keeping the inputs to the computation (e.g., patient data) secret. But SMPC can have high memory overhead, making it difficult to scale such applications to large problem sizes. As a result, while expressive cryptography has seen some notable real-world usage, such as Meta using SMPC in its advertising business, existing adoption is not widespread, limited to incipient and isolated deployments.

This dissertation studies how to design and build networked systems to enable expressive cryptography to reach its full transformative potential. We present six system design techniques for systems relating to expressive cryptography, classified into two high-level approaches. We validate our techniques by using them to design and implement four systems: MAGE, *TCPlp*, JEDI, and Ghostor.

Our first high-level approach is to make expressive cryptography generically more efficient by redesigning the underlying systems that expressive cryptography uses. For example, MAGE provides virtual memory for SMPC and FHE at nearly zero cost, allowing them to efficiently scale beyond the available memory to larger problem sizes. *TCPlp* is a performant TCP-based transport layer for low-power wireless networks, which allows the large ciphertexts and signatures associated with expressive cryptography to be efficiently transferred over the network.

Our second high-level approach is to make expressive cryptography practical for particular applications by rethinking how and when to use expressive cryptography. For example, we designed Ghostor, a data-sharing system, and JEDI, an end-to-end encryption protocol for publish-subscribe

IoT deployments, using this approach. Ghostor uses a blockchain and JEDI leverages policy-based encryption, but they are carefully designed to use these components *rarely* and *outside of the critical path* of user-facing operations.

We further validate our techniques by using them to analyze related work, to identify existing work that applies our techniques and opportunities to improve existing systems using our techniques. Then, we discuss the impact of our work, including the adoption of *TCPlp* as the TCP implementation in OpenThread, an open-source network stack used in the smart home IoT industry, including by Amazon Eero and Google Nest. We hope that our techniques, and the systems we designed using them, will accelerate the widespread adoption of expressive cryptography, bringing stronger security to existing applications and enabling exciting new ones.

To my teachers.

Contents

Contents **ii**

List of Figures **viii**

List of Tables **xi**

1 Introduction **1**
1.1 Motivation: Expressive Cryptography and its Potential 1
1.2 Problem: Systems Built on Expressive Cryptography are Inefficient 3
1.3 Our Approach to Designing Systems for Expressive Cryptography 3
1.4 Systems We Built 4
1.4.1 MAGE [295] 5
1.4.2 *TCPlp* [292, 293, 278] 5
1.4.3 Ghostor [226] 6
1.4.4 JEDI [296] 6
1.5 Thesis Statement and Roadmap for This Dissertation 7

2 Background **8**
2.1 Expressive Cryptography 8
2.1.1 Expressive Cryptography for Protecting Confidentiality 9
2.1.2 Expressive Cryptography for Protecting Integrity 11
2.1.3 Expressive Cryptography for Protecting Computation 12
2.2 Efficiency and Overheads of Expressive Cryptography 15
2.2.1 Types of Cryptographic Overhead 15
2.2.2 Trade-Off Between Expressivity and Efficiency 18
2.3 Techniques for Making Expressive Cryptography Efficient 18
2.3.1 Generic Theoretical Improvements 18
2.3.2 Specialized Cryptographic Schemes 19
2.3.3 Systems Techniques 19
2.4 Conclusion and Thesis Statement Revisited 20

3 System Design Techniques for Expressive Cryptography **22**

3.1 Designing Systems that Support Expressive Cryptography . . . 22
3.1.1 Manage Resources According to the Structure of the Computation . . . 23
3.1.2 Identify the Bottleneck and Generically Optimize It . . . 24
3.2 Designing Systems that Use Expressive Cryptography . . . 24
3.2.1 Use Expressive Cryptography Rarely and Off of the Critical Path . . . 25
3.2.2 Make the Frequency of Expressive Cryptographic Operations Tunable . . . 26
3.2.3 Identify and Use the Cheapest Cryptographic Primitive . . . 27
3.2.4 Develop Application-Specific Cryptographic Interfaces . . . 28
3.3 When to Use Each Class of Techniques . . . 29

4 Supporting Secure Computation with Nearly Zero-Cost Virtual Memory 32
4.1 Introduction . . . 32
4.2 Secure Computation Background . . . 35
4.2.1 Circuit Representation . . . 35
4.2.2 CKKS Homomorphic Encryption . . . 35
4.2.3 Garbled Circuits . . . 36
4.2.4 Efficiently Executing Circuits . . . 36
4.3 Memory Overhead of Secure Computation . . . 37
4.3.1 Analysis of the Memory Demand . . . 37
4.3.2 Scaling Collaborative Applications . . . 38
4.4 System Overview . . . 38
4.4.1 Address Translation . . . 39
4.4.2 Bytecode Representation . . . 40
4.4.3 Ecosystem and Extensibility . . . 41
4.5 Engine . . . 42
4.5.1 Parallel/Distributed Engine . . . 42
4.5.2 Distributed SMPC . . . 43
4.6 Planner . . . 44
4.6.1 Organization of the Planner . . . 44
4.6.2 First Stage: Placement . . . 45
4.6.3 Second Stage: Replacement . . . 47
4.6.4 Third Stage: Scheduling . . . 47
4.7 Implementation . . . 48
4.7.1 Interpreter . . . 49
4.7.2 Extending MAGE with New Protocols . . . 49
4.7.3 Garbled Circuit Protocol Driver . . . 50
4.7.4 CKKS Protocol Driver . . . 50
4.8 Evaluation . . . 50
4.8.1 Workloads . . . 50
4.8.2 Empirical Methodology . . . 52
4.8.3 Comparison to Existing Frameworks . . . 53
4.8.4 Overhead of Swapping Pages . . . 53

4.8.5 Overhead of Planning . . . 55
4.8.6 Impact of Parallelism . . . 56
4.8.7 SMPC in Wide-Area Networks . . . 57
4.8.8 Applications . . . 58
4.9 Related Work . . . 60
4.10 Conclusion . . . 61

5 Supporting Cryptography in Low-Power Wireless Systems with Performant TCP 62
5.1 Introduction . . . 63
5.2 Background and Related Work . . . 65
5.2.1 Low-Power and Lossy Networks (LLNs) . . . 65
5.2.2 TCP/IP for Embedded LLN-Class Devices . . . 67
5.3 Motivation . . . 68
5.3.1 The Case for TCP in LLNs . . . 68
5.3.2 Anemometry: An Example TCP-Based LLN Application . . . 69
5.4 Empirical Methodology . . . 70
5.4.1 Network Stack . . . 70
5.4.2 Embedded Hardware . . . 71
5.5 Implementation of *TCPlp* . . . 72
5.5.1 Supported TCP Features . . . 73
5.5.2 Concurrency Model . . . 74
5.5.3 Timer Event Management . . . 75
5.5.4 Connection State for *TCPlp* . . . 76
5.5.5 Memory-Efficient Data Buffering . . . 76
5.6 TCP in a Low-Power Network . . . 78
5.6.1 Reducing Header Overhead using MSS . . . 78
5.6.2 Impact of Buffer Size . . . 79
5.6.3 Direct TCP Connection . . . 80
5.6.4 Upper Bound on Single-Hop Goodput . . . 81
5.7 TCP Over Multiple Wireless Hops . . . 82
5.7.1 Mitigating Hidden Terminals in LLNs . . . 82
5.7.2 Upper Bound on Multi-Hop Goodput . . . 84
5.7.3 TCP Congestion Control in LLNs . . . 84
5.7.4 Modeling TCP Goodput in an LLN . . . 85
5.8 TCP in LLN Applications . . . 87
5.8.1 Web Server Application Scenario . . . 88
5.8.2 Sense-and-Send Application Scenario . . . 90
5.8.3 Event Detection Application Scenario . . . 95
5.9 Conclusion . . . 95
5.9.1 Implications for Applications Relating to Cryptography . . . 96
5.9.2 Broader Implications for Networking . . . 97

6 Using Cryptography Efficiently for Anonymous and Verifiable Data Sharing 98
6.1 Introduction . . . 98
6.1.1 Hiding User Identities . . . 101
6.1.2 Verifiable Consistency . . . 102
6.1.3 Summary of Contributions . . . 103
6.2 System Overview . . . 104
6.3 Threat Model and Security Guarantees . . . 106
6.3.1 Assumptions . . . 106
6.3.2 Verifiable Linearizability . . . 106
6.3.3 Anonymity . . . 107
6.4 Hiding User Identities . . . 107
6.4.1 No User Login or User-Specific Mailboxes . . . 108
6.4.2 No Server-Visible ACLs . . . 108
6.4.3 No Server-Visible User Public Keys . . . 109
6.4.4 No Client-Side Caching . . . 110
6.4.5 Careful Application Design . . . 110
6.5 Achieving Verifiable Consistency . . . 110
6.5.1 Hash Chain of Digests . . . 110
6.5.2 Checkpoint and Verification . . . 111
6.5.3 Multiple Objects per Checkpoint . . . 112
6.5.4 Concurrent Operations on a Single Object . . . 112
6.6 Mitigating Resource Abuse . . . 114
6.6.1 Anonymous Payments . . . 114
6.6.2 Proof of Work (PoW) . . . 114
6.6.3 Using Anonymous Payments and Proof of Work Together . . . 115
6.7 Full Protocol Description . . . 115
6.7.1 `GET` Protocol . . . 115
6.7.2 `PUT` Protocol . . . 115
6.7.3 Access Control . . . 116
6.7.4 Object Creation . . . 117
6.7.5 Verification Procedure . . . 117
6.7.6 Payment . . . 118
6.8 Applying Ghostor to Applications . . . 118
6.9 Implementation . . . 119
6.10 Evaluation . . . 120
6.10.1 Microbenchmarks . . . 120
6.10.2 Server-Side Overhead . . . 121
6.10.3 End-to-End Latency . . . 124
6.10.4 Zcash . . . 125
6.11 Extensions . . . 126
6.11.1 Files and Directories . . . 126
6.11.2 Scalability . . . 127

6.12 Related Work . . . 127
6.13 Conclusion . . . 129

7 Using Cryptography Efficiently for Many-to-Many End-to-End Encryption for IoT 130
7.1 Introduction . . . 130
7.1.1 Requirements for JEDI . . . 131
7.1.2 Overview of JEDI . . . 133
7.1.3 Summary of Evaluation . . . 136
7.2 System Model and Threat Model . . . 136
7.2.1 Trust Assumptions . . . 137
7.2.2 Applying JEDI to an Existing System . . . 137
7.2.3 Comparison to a Naïve Key Server Model . . . 138
7.2.4 IoT Gateways . . . 138
7.2.5 Generalizability of JEDI's Model . . . 139
7.2.6 Security Goals . . . 139
7.3 End-to-End Encryption . . . 139
7.3.1 Building Block: WKD-IBE . . . 140
7.3.2 Concurrent Hierarchies in JEDI . . . 142
7.3.3 Overview of Encryption in JEDI . . . 142
7.3.4 Expressing URI/Time as a Pattern . . . 143
7.3.5 Producing a Key Set for Delegation . . . 143
7.3.6 Using WKD-IBE Efficiently . . . 144
7.3.7 Revocation . . . 146
7.3.8 Simple Solution: Revocation via Expiry . . . 146
7.3.9 Immediate Revocation (Extended Paper) . . . 146
7.3.10 Extensions . . . 146
7.3.11 Security Guarantee . . . 147
7.4 Integrity . . . 151
7.4.1 Starting Solution: Signature Chains . . . 151
7.4.2 Anonymous Signatures . . . 151
7.4.3 Using WKD-IBE for Signatures Efficiently . . . 153
7.4.4 Security Guarantee . . . 155
7.5 Implementation . . . 156
7.5.1 C/C++ Cryptography Library . . . 157
7.5.2 Application of JEDI to bw2 . . . 157
7.6 Evaluation . . . 158
7.6.1 Building Block Comparison: HIBE, WKD-IBE, and KP-ABE . . . 158
7.6.2 Microbenchmarks . . . 162
7.6.3 Performance of JEDI in bw2 . . . 162
7.6.4 Feasibility on Ultra Low-Power Devices . . . 165
7.6.5 Comparison to Other Systems . . . 167
7.7 Related Work . . . 170

7.8 Conclusion . . . 171

8 Related Work **173**
8.1 Other Systems that Exemplify Our System Design Principles . . . 173
8.1.1 Messaging and Storage Systems . . . 173
8.1.2 Cryptographic Planners . . . 174
8.1.3 Performance-Oriented Systems . . . 176
8.2 Applying Our System Design Principles to Other Systems . . . 176
8.2.1 Applicability of Techniques for Supporting Expressive Cryptography . . . 176
8.2.2 Applicability of Techniques for Using Expressive Cryptography . . . 177

9 Conclusion **179**
9.1 Impact . . . 179
9.1.1 Integration of *TCPlp* into Thread and OpenThread . . . 179
9.1.2 Integration of JEDI into WAVE and WAVEMQ . . . 180
9.2 Future Research Directions . . . 180
9.2.1 Future Systems that Support Expressive Cryptography . . . 180
9.2.2 Future Systems that Use Expressive Cryptography . . . 181
9.3 Summary . . . 182

Bibliography **184**

A Ghostor's Security Guarantees **217**
A.1 Ghostor's Privacy Guarantee . . . 217
A.1.1 Overview . . . 217
A.1.2 Real World . . . 220
A.1.3 Ideal World . . . 221
A.1.4 Security Definition . . . 223
A.1.5 Proof of Ghostor's Privacy . . . 224
A.2 Ghostor's Integrity Guarantee . . . 230
A.2.1 Linearizability . . . 230
A.2.2 Verifiable Linearizability . . . 231

List of Figures

3.1 Classical system design: the system provides applications with access to the hardware. 29
3.2 With expressive cryptography, there are two system layers: one providing applications with access to expressive cryptography, and another providing expressive cryptography with access to the hardware. 30

4.1 Overview of MAGE. It consists of two phases: planning (top) and execution (bottom). . 38
4.2 MAGE's envisioned ecosystem, with planning as the narrow waist. 41
4.3 Example of distributed SMPC with MAGE. Workers are denoted as circles with W. Solid lines indicate connections managed by MAGE's engine; dashed lines indicate connections managed by the protocol driver. 43
4.4 MAGE's planner's workflow, with its three stages. 44
4.5 Example code in an Integer-based DSL internal to C++ to solve Yao's Millionaire's problem. 46
4.6 Comparison of MAGE and EMP-toolkit. 54
4.7 Comparison of MAGE and SEAL. 54
4.8 Performance of Unbounded, OS Swapping, and MAGE, normalized by the time for Unbounded; absolute times, in seconds, are printed at the upper left corner of each bar. 55
4.9 Repeat of Figure 4.8, with larger problem sizes and a 16 GiB memory limit (note the larger y-axis scale). 55
4.10 Normalized performance of Unbounded, OS Swapping, and MAGE, parallelized over $p = 4$ workers (per party). 56
4.11 Wide-area garbled circuit performance in MAGE. 57
4.12 Scaling password reuse detection with MAGE. 59
4.13 Scaling computational PIR with MAGE. 59

5.1 Hamilton-based ultrasonic anemometer. 69
5.2 Snapshot of uplink routes in OpenThread topology at transmission power of -8 dBm (5 hops). Node 1 is the border router with Internet connectivity. 72
5.3 Naïve and final TCP receive buffers. 78
5.4 TCP goodput over one IEEE 802.15.4 hop. 80
5.5 Analysis of overhead limiting *TCPlp*'s goodput. 81

5.6 Effect of varying time between link-layer retransmissions. Reported "segment loss" is the loss rate of TCP segments, not individual IEEE 802.15.4 frames. It includes only losses not masked by link-layer retries. 83
5.7 Congestion behavior of TCP over IEEE 802.15.4. 85
5.8 Latency of web request: CoAP vs. HTTP/TCP. 88
5.9 Goodput: CoAP vs. HTTP/TCP. 89
5.10 Effect of batching on power consumption. 91
5.11 Performance with injected packet loss. 92
5.12 Radio duty cycle of TCP and CoAP in a lossy wireless environment, in one representative trial (losses are caused by natural human activity). 94
5.13 CoAP, CoCoA, and TCP with four competing flows. 96

6.1 An example of what a server attacker sees in a typical end-to-end encrypted (E2EE) system versus Ghostor's Anonymous E2EE. 100
6.2 Information leakage in a data-sharing system and associated privacy properties. 100
6.3 Ghostor's contributions. Ghostor's techniques can be applied to both oblivious and non-oblivious systems. 101
6.4 System overview of Ghostor. Shaded areas indicate components introduced by Ghostor. 105
6.5 Object layout in Ghostor. 109
6.6 Blind signature. 121
6.7 YCSB workloads (R: read, W: write). 121
6.8 Operations for verification. 121
6.9 Latency measurements. 123
6.10 Benchmarks comparing throughput of the six setups described in Section 6.10.2. 124
6.11 Microbenchmarks of PoW mechanism and Tor. 125
6.12 End-to-end latencies of client-side operations. 126

7.1 IoT comprises diverse devices that span more than four orders of magnitude of computing power (estimated in Dhrystone MIPS).[1] 132
7.2 JEDI keys can be qualified and delegated, supporting decentralized, cryptographically-enforced access control via key delegation. Each person has a decryption key for the indicated resource subtree that is valid until the indicated expiry time. Black arrows denote delegation. 133
7.3 Applying JEDI to a smart buildings IoT system. Components introduced by JEDI are shaded. The subscriber's key is obtained via JEDI's decentralized delegation (Figure 7.2). 137
7.4 Pattern *S* used to encrypt message sent to `a/b` on June 08, 2017 at 6 AM. The figure uses 8 slots for space reasons; JEDI is meant to be used with more slots (e.g., 20). . . . 143
7.5 Pattern *S* for a private key granting access to `a/+/b/*` at 8 AM every day. The figure uses 8 slots for space reasons; JEDI is meant to be used with more slots (e.g., 20). . . . 147
7.6 Latency of **Encrypt**, **Decrypt**, **KeyDer**, **Sign**, and **Verify** with 20 attributes. 162
7.7 Critical-path operations in bw2, with/without JEDI. 163
7.8 Occasional bw2 operations, with and without JEDI. 164

A.1 Overview of Real World and Ideal World. 219

List of Tables

4.1 Planning times (s) and peak memory use of the planner (MiB) for workloads in Figure 4.8 and Figure 4.9. . . . 56

5.1 Impact of techniques to run full-scale TCP in LLNs. . . . 64
5.2 Comparison of the platforms we used (Hamilton and Firestorm) to TelosB and Raspberry Pi. . . . 71
5.3 Comparison of features among embedded TCP stacks: uIP (Contiki), BLIP (TinyOS), GNRC (RIOT), and *TCPlp (our work)*. . . . 73
5.4 Memory usage of *TCPlp* on TinyOS. Our *TCPlp* implementation spans three modules: (1) protocol implementation, (2) event scheduler that injects callbacks into userspace, and (3) user library. . . . 76
5.5 Memory usage of *TCPlp* on RIOT OS. We also include RIOT's `posix_sockets` module, used by *TCPlp* to provide a Unix-like interface. . . . 77
5.6 Comparison of TCP/IP links. . . . 78
5.7 Header overhead with 6LoWPAN fragmentation. . . . 78
5.8 Comparison of *TCPlp* to existing TCP implementations used in network studies over IEEE 802.15.4 networks.[2] Goodput figures obtained by reading graphs in the original paper (rather than stated numbers) are marked with the $\approx$ symbol. . . . 82
5.9 Performance in the testbed over a full day, averaged over multiple trials. The ideal protocol (Section 5.8.2.2) would have a radio duty cycle of $\approx$ 0.63%–0.70% under similarly lossy conditions. . . . 94

6.1 Our goals and how Ghostor achieves each one. . . . 103
6.2 Per-object keys in Ghostor. The server uses the global signing keypair $(\mathsf{SVK}, \mathsf{SSK})$ to sign digests for objects. . . . 108
6.3 A digest for an operation in Ghostor. . . . 111

7.1 Performance comparison of HIBE, WKD-IBE, and KP-ABE in terms of pairings and exponentiations. We omit operations that can be precomputed once for all IDs (attribute sets) in the HIBE/WKD-IBE/KP-ABE system. **KeyDer^1** indicates deriving the new key from the master key, and **KeyDer^2** indicates the other case. . . . 160

7.2 Size comparison of HIBE, WKD-IBE, and KP-ABE in terms of number of group elements. For elliptic curves that we used, elements of $\mathbb{G}_1$ are 48 B each, elements of $\mathbb{G}_2$ are 96 B each, and elements of $\mathbb{G}_T$ are 576 B each. 161
7.3 Latency of JEDI's implementation of BLS12-381. 161
7.4 CPU and power costs on the Hamilton platform. 165
7.5 Average current and expected battery life (for 1400 mAh battery) for sense-and-send, with varying sample interval. 166
7.6 Comparison of JEDI with other crypto-based IoT/cloud systems. 168

Co-Authored Material

Parts of this dissertation are based on previously published material co-authored with others, as follows.

- Chapter 4 is based on the following publication [295]:
  Sam Kumar, David E. Culler, and Raluca Ada Popa. "MAGE: Nearly Zero-Cost Virtual Memory for Secure Computation". In: *OSDI*. USENIX, 2021.
- Chapter 5 is based on the following publication [293]:
  Sam Kumar, Michael P Andersen, Hyung-Sin Kim, and David E. Culler. "Performant TCP for Low-Power Wireless Networks". In: *NSDI*. USENIX, 2020.
- Chapter 6 is based on the following publication [226]:
  Yuncong Hu, Sam Kumar, and Raluca Ada Popa. "Ghostor: Toward a Secure Data-Sharing System from Decentralized Trust". In: *NSDI*. USENIX, 2020.
- Chapter 7 is based on the following publication [296]:
  Sam Kumar, Yuncong Hu, Michael P Andersen, Raluca Ada Popa, and David E. Culler. "JEDI: Many-to-Many End-to-End Encryption and Key Delegation for IoT". in: *USENIX Security*. USENIX, 2019.

Acknowledgments

My ten years at UC Berkeley have been a tremendous journey, and a huge part of that journey is the people I met along the way. Having arrived at Berkeley knowing little about computer science and even less about academic research, I am particularly grateful to those individuals who influenced my outlook on computer science, research, and academia. It is for this reason that I am dedicating this dissertation to my *teachers*, broadly defined—those who shared their knowledge, wisdom, and advice with me, shaping my perspectives and guiding me to where I am today.

I am deeply grateful to David E. Culler, my PhD co-advisor. He has been an excellent mentor to me throughout my time at UC Berkeley. In my first year as an undergraduate, David took a chance on me, giving me the opportunity to work in his research group even though I had no prior research experience. I enjoyed working with him so much that I continued at UC Berkeley as his student in graduate school! Over the years, David taught me how to do research and instilled in me a deep appreciation for what it means to do science. In research, he gave me guidance and support but also the flexibility to follow my own ideas. I appreciate now, more than I ever did before, David's advice to consciously focus on my *education* while in graduate school.

I am equally grateful to Raluca Ada Popa, my other PhD co-advisor. When my research drifted into security in my early years of graduate school, Raluca was willing to collaborate with me and ultimately co-advise me even though I had no prior research experience in security or cryptography. She is an excellent collaborator and I learned tremendously from her guidance at each part of the research process—formulating ideas, developing them into projects, and technical writing and presentation. Her feedback and support pushed me to improve in all of these areas, and I am a much better researcher for it.

I am also deeply thankful to Scott Shenker, a member of my dissertation committee and a wonderful collaborator. In Fall 2020, when David's other students had all graduated and David's research group had come to an end, Scott graciously welcomed me into the NetSys Lab, which became my new systems home at UC Berkeley. Although I was not his student, Scott spent a great deal of time to mentor me and give me advice. I deeply admire the clarity of his thought, the directness of his advice, his principled approach to research, and his humility and warmth as an individual. I only wish I had reached out to Scott and started working with him sooner!

I am fortunate to have also interacted with numerous other faculty members. Natacha Crooks, Prabal Dutta, Dan Garcia, Joey Gonzalez, John Kubiatowicz, Amy Ousterhout, Aurojit Panda, Sylvia Ratnasamy, Ion Stoica, and Matei Zaharia all gave me invaluable career advice during my final year of graduate school. I am also grateful to have had the opportunity to engage with some of them in research, through either collaboration or receiving their technical advice. I would like to thank John Kubiatowicz, whose CS 162 course I took as an undergraduate in Fall 2015 helped me discover my interest in computer systems. Chris Hoofnagle helped me explore the broader implications of cybersecurity and is a member of my dissertation committee. Stuart Cheshire is an engineer at Apple, not a faculty member, but his mentorship and support, both during my internship at Apple and afterward, were essential for achieving real-world adoption and impact for *TCPlp*.

Staff members, both in the research labs I worked in and in the EECS department, worked diligently and professionally to make my graduate school experience as smooth as possible. I

would particularly like to thank Albert Goto for the many late nights he spent helping me with the networking testbed for the *TCPlp* project and for his generosity, kindness, and support as a friend.

I am also thankful to my colleagues in the EECS department. I started out working primarily in 410 Soda, sharing the lab space with a close-knit group of graduate students and postdocs: Moustafa AbdelBaky, Michael Andersen, Kaifei Chen, Gabe Fierro, Hyung-Sin Kim, and Jack Kolb. The stimulating environment in 410 Soda, particularly my office mates' enthusiasm for system building, was tremendously influential for me. I also got to interact with many other colleagues, including Emmanuel Amaro, Chris Branner-Augmon, Lloyd Brown, Weikeng Chen, Audrey Cheng, David Chu, Emma Dauterman, Tess Despres, Chris Douglas, Lisa Dunlap, Vivian Fang, Silvery Fu, Narek Galstyan, Rolando Garcia, Yuncong Hu, Rishabh Iyer, Paras Jain, Alex Krentsel, Sukrit Kalra, Darya Kaviani, Shadaj Laddad, Shu Liu, Zhihong Luo, Emily Marx, Sarah McClure, Mae Milano, Pratyush Mishra, Norman Mu, Micah Murray, Akshay Narayan, Amy Ousterhout, Charles Packer, Ashwinee Panda, Shishir Patil, Julien Piet, Rishabh Poddar, Conor Power, Deevashwer Rathee, Mayank Rathee, Daniel Rothchild, Peter Schafhalter, Kalyanaraman Shankari, Katerina Sotiraki, Sijun Tan, Mark Theis, Tenzin Ukyab, Stephanie Wang, Jean-Luc Watson, Justin Wong, Sarah Wooders, Samyu Yagati, Alice Yeh, Wen Zhang, Wenting Zheng, and other wonderful folks. Together, they made Berkeley EECS the remarkable environment that it is. I am fortunate to have had the opportunity to collaborate with some of them on research projects, and I will look back fondly on the time I could spend with some of them outside of research.

In Fall 2019, I had a wonderful time working with Will Wang when we were the head teaching assistants for CS 162. As the instructor for CS 162 in Summer 2020, I got to work with a wonderful group of teaching assistants and readers—Bobby Yan, William Hsu, Jonathan Shi, Kevin Yu, Ganeshkumar Ashokavardanan, Gillian Chu, and Sean Huang—who kept my course running smoothly even though classes had just moved online due to the COVID-19 pandemic. I am thankful (once again!) to David Culler and Dan Garcia, who supported me in my teaching interests and whose advice and guidance helped make these experiences both successful and enjoyable for me.

Finally, I would like to thank my parents and brother for their continuous support and unconditional love. They have shaped who I am both before and during my time at Berkeley, and I owe my accomplishments in graduate school, including this dissertation, to them.

***

I am fortunate to have been supported by a Berkeley Fellowship for Graduate Study and by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE-1752814. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. The work in this dissertation was also supported by Intel/NSF CPS-Security Grants 1505773 and 20153754, NSF CPS Grant 1239552, Department of Energy Grant DE-EE0007685, California Energy Commission Grant EPC-15-057, NSF CISE Expeditions Award CCF-1730628, NSF CAREER 1943347, and gifts from the Sloan Foundation, Bakar Fellows Program, Alibaba, Amazon Web Services, Ant Group, Ericsson, Facebook, Futurewei, Google, Intel, Microsoft, Nvidia, Scotiabank, Splunk, and VMware.

Chapter 1

Introduction

This dissertation shows how to realize the potential of expressive cryptography by building efficient computer systems. In this chapter, we explain what expressive cryptography is, its potential to enable transformative applications, its high cost as an obstacle to enabling those applications, and our approach of using system design to overcome the high cost of expressive cryptography. Then, we briefly summarize the systems we built that we will describe in this dissertation and provide a roadmap for the chapters that follow.

1.1 Motivation: Expressive Cryptography and its Potential

Expressive cryptography can be understood in contrast to public-key cryptography. For example, public-key cryptography guarantees that information encrypted with a public key is only readable by parties with the corresponding secret key. **Expressive cryptography** provides more expressive control over *who* can access *what* information [82, 209]. Secure Multi-Party Computation (SMPC), Fully Homomorphic Encryption (FHE), policy-based encryption, and the protocols behind blockchains are all examples of expressive cryptography. For example, certain policy-based encryption schemes can allow anyone with particular attributes (instead of a single secret key) to decrypt a message, and SMPC and FHE allow computation on encrypted data, revealing a function of the input data rather than the original message directly.

To understand the potential of expressive cryptography, public-key cryptography is an apt analogy. When public-key cryptography was first widely adopted in the 1980s and 1990s, it had a revolutionary impact on computing, enabling new applications that handle sensitive data. For example, applications like e-commerce (e.g., Amazon, eBay), patient portals and telemedicine (e.g., Epic, Teladoc), and end-to-end encrypted messaging (e.g., Signal, WhatsApp) depend fundamentally on public-key cryptography to protect payment information, patient data, and users' messages, and they are regularly and widely used in society today.

Expressive cryptography has the potential to be as transformative as public-key cryptography. The reason is that expressive cryptography can solve important problems that public-key cryptography cannot. For example, consider the problem in which multiple hospitals each have patient

datasets and would like to perform research using the combined dataset. Due to legal regulations and privacy concerns, the hospitals cannot directly share patient data. Public-key cryptography, on its own, does not help with this; a hospital can encrypt its patients' data with public-key cryptography and give another hospital the key to decrypt its data, but this is tantamount to sharing the data outright. The hospitals need to share patient data such that they can be used *for the particular computation* needed in the hospitals' research, but not in any other way.

A form of expressive cryptography, **SMPC**, enables exactly that. It enables multiple parties to run a function f (which all parties agree on) on their combined dataset, while guaranteeing that each party learns nothing about the other parties' inputs except for the output of the function f. This not only solves the problem above regarding collaboration among hospitals, but a larger class of problems [196], which we call *secure collaborative data analytics*. For example, multiple competing banks cannot directly share their customers' transaction data with one another, but they may need to combine their transaction datasets to look for fraudulent transaction patterns across the banks (e.g., money laundering); SMPC could allow them to detect fraud while keeping their customers' transaction data private. In fact, the chief risk officer of Scotiabank stated that "collaboration will be vital" for anti-money-laundering and that the "ability to put together our data sets and collaborate on typologies of attack—and the use of both advanced-encryption methods and analytics methods to mine the data—will enhance yields by orders of magnitude," [161] indicating industry interest in such solutions. As another example, researchers at Boston University collaborated with several organizations to apply SMPC to societal problems, such as addressing the gender wage gap and ensuring economic inclusion of minority-owned businesses [386]. SMPC helps to tackle these societal problems by enabling companies to combine their private datasets (e.g., employee salaries and corporate spending patterns), which are too sensitive to share directly, for analysis to track progress toward addressing these societal issues. Separately, in the advertising space, SMPC allows an advertiser to measure the effectiveness of an ad campaign in a privacy-preserving way—for example, the parties may learn how many ad views resulted in customer acquisitions without the advertiser learning who viewed the ad and the advertising agency (e.g., Google or Meta) learning who the advertiser's customers are [431]. This application of SMPC is seeing industry deployment; Meta offers a product for it, namely Private Lift [396], and Google has open-sourced an SMPC tool, Private Join and Compute [464], designed with this application in mind [243]. Other industry applications of SMPC abound; Fireblocks provides SMPC-based wallets used to secure digital assets [172, 173], and an industry consortium, the MPC Alliance [351], has formed to promote SMPC-based technologies.

Other kinds of expressive cryptography also enable compelling new applications, though for brevity we do not explain them in as much detail as we did for SMPC. Just as end-to-end encryption is widely deployed in chat applications like WhatsApp and Signal, **policy-based encryption** schemes could bring end-to-end encryption to data lakes and the emerging Internet of Things (IoT) [197]. The cryptographic **protocols behind blockchains** could transform finance, supply chains, and healthcare [286]. Given the potential impact of expressive cryptography, its widespread adoption could be transformative for society.

1.2 Problem: Systems Built on Expressive Cryptography are Inefficient

Despite its enormous potential, expressive cryptography has not seen widespread use. While technologies like SMPC have enormous potential and are seeing intense industry interest, actual deployments remain incipient and isolated. A main reason is that expressive cryptography is usually much more expensive than widely deployed cryptography like public-key cryptography. In this case, "expensive" means that the tools are slower than public-key cryptography or that they consume more computing resources (e.g., more CPU time, more memory, more network bandwidth, etc.). As a result, integrating expressive cryptography into real-world systems may incur unacceptable performance overheads.

We can see the impact of this in the expressive cryptographic tools that we have discussed. SMPC is used in *point solutions* custom-tailored and hand-built by expert cryptographers [127, 260], but have not resulted in widespread *generic* computation on encrypted data due to their large cost [431]. For example, Google's Private Join and Compute tool uses a *specialized* SMPC tool for a particular class of problems called "private set intersection sum with cardinality" [243], likely because using an SMPC tool for generic computation is almost always much more expensive. Policy-based encryption has been hailed as potentially transformative for IoT [197], yet its energy cost can be prohibitive for low-power embedded sensors that are part of IoT. And while blockchains have undeniably impacted digital currency and payment systems, their impact in other areas that could benefit from public verifiability, like data storage, has been comparatively limited, due to the cost of performing a blockchain transaction for each data update.

The work presented in this dissertation addresses this problem by rethinking computer system design for expressive cryptography. For example, it enables SMPC to scale to large problem sizes despite its memory overhead (Chapter 4), high-throughput data-sharing systems to use a blockchain for verifiability despite its transaction overheads (Chapter 6), and ultra low-power IoT devices to benefit from policy-based encryption despite its energy overhead (Chapter 7).

1.3 Our Approach to Designing Systems for Expressive Cryptography

In the research presented in this dissertation, *we design and build networked systems that allow expressive cryptography to reach its potential.* We believe that this will enable computer users to benefit from the stronger security (e.g., end-to-end encryption for IoT communication) and better functionality (e.g., fine-grained access control for sensitive data) afforded by expressive cryptography. Two high-level approaches pervade the work in this dissertation.

First, we can make expressive cryptography *generically* more efficient by redesigning the underlying systems that expressive cryptography uses. For example, MAGE [295] observes that cryptographic protocols like SMPC and FHE have a special structure called *obliviousness* and rethinks memory management accordingly. This improves the performance of data-intensive SM-

PC/FHE programs by up to an order of magnitude compared to the operating system's default memory management. Sometimes, the gains apply beyond expressive cryptography. For example, *TCPlp* [293], a performant TCP-based transport layer for low-power wireless networks, allows the large ciphertexts and signatures associated with policy-based cryptography to be efficiently sent over extremely resource-constrained network. Yet *TCPlp* broadly benefits IoT, separate from cryptography—it enables direct and gateway-free Internet connectivity for IoT devices, making them first-class citizens of the Internet.

Second, we can efficiently secure systems with expressive cryptography by rethinking how and when they use the inherently expensive cryptographic components. One way we do this is to invoke expressive cryptography *outside of the critical path* of user-facing operations. For example, JEDI [296] leverages policy-based encryption and Ghostor [226] uses a blockchain, but they are carefully designed to use these components in the background—not in the critical path of sending/receiving data in JEDI or accessing/sharing data in Ghostor. To control the total cost of using expressive cryptography, we tie expressive cryptographic operations to *tunable aspects* of the system's functionality—for example, the granularity at which JEDI expiry times may be specified and the delay after which Ghostor may detect an integrity violation. This allows for tuning cryptographic costs to the application at hand (e.g., energy budget of an IoT device).

Our approach, which focuses on **system design**, differs from prior research. Most prior efforts focus on **cryptographic design**, improving the underlying math [398] or specializing it to the application [127, 260]. The two approaches are complementary—our systems would benefit from cryptographic improvements. In some cases, they are actually highly synergistic. For example, computational improvements to SMPC and FHE would increase their memory intensity, making MAGE's techniques even more relevant. We focus on system design for two reasons. First, cryptographic design has inherent limits. For example, cryptographers made great strides in making generic SMPC more efficient [398], but such improvements have begun to plateau. For example, lower bounds on communication in certain SMPC protocols have already been attained [504]. Furthermore, building point solutions—cryptographic protocols tailored to an application or a particular class of applications—can be effective for certain applications, but is not necessarily possible for all applications. Second, the theory has advanced so far, in some cases, that the applications based on expressive cryptography are within striking distance of practicality [209]. In such cases, significant costs may stem from systems-related inefficiencies rather than from the cryptographic protocols being fundamentally slow. MAGE, for example, is based on the observation that significant costs for large SMPC and FHE workloads stem from memory management in the underlying system.

1.4 Systems We Built

We demonstrate the validity of the approaches outlined above through the design and implementation of several systems. This section briefly describes each system; later chapters in this dissertation explain them in greater depth.

1.4.1 MAGE [295]

Secure computation (SC) protocols, like SMPC and FHE, allow *computation on encrypted data*. Unfortunately, SC often does not scale to large problem sizes. For example, prior research works that use SMPC to solve data analytics problems have found that SMPC "in practice only scales to a few thousand input records" [462]. The reason is that, for large computations and large inputs, SC protocols quickly run out of memory and become prohibitively slow due to the overhead of swapping to secondary storage.

MAGE is an execution engine for SC that mitigates this problem. Our key observation is that **SC protocols have a property called *obliviousness* that opens new opportunities for managing memory.** That SC protocols are oblivious means that their memory access patterns are independent of the program's inputs. This is necessary because of SC protocols' security guarantees; otherwise, a party executing SC could learn about the inputs by observing how it accesses memory. Because an SC program is oblivious, MAGE can compute its memory access pattern in advance and use it to preplan memory management. In this paradigm, which we call *memory programming*, MAGE can make better policy decisions than the operating system. First, while the operating system must use heuristics to decide which page to evict on a page fault, MAGE directly uses MIN, Belady's optimal paging algorithm. Second, MAGE *prefetches* according to the access pattern, with no false positives or false negatives.

MAGE outperforms Linux by up to an order of magnitude. Despite the costs of swapping memory, MAGE runs SC programs at nearly the same speed as if they had *unbounded* memory to fit the entire computation, providing virtual memory at nearly zero cost. This makes it easier to scale data-intensive SC workloads, such as secure collaborative data analytics, to large, real-world datasets.

1.4.2 *TCPlp* [292, 293, 278]

Ultra low-power IoT devices and networked sensors typically use low-power and lossy networks (LLNs), like IEEE 802.15.4, rather than Wi-Fi. Since LLN research began, TCP has generally been considered unsuitable for LLNs. As a result, standard LLN network stacks either do not support TCP, or provide simplified TCP implementations that perform poorly. This is an obstacle for expressive cryptography, as it is difficult to transfer large keys/ciphertexts without TCP. Yet the lack of TCP has farther-reaching consequences. Because LLN devices often do not run TCP, communication with external TCP/IP-based services requires *application-layer gateways*. As a result, IoT applications tend to develop as vertically integrated silos, with little to no **interoperability** across ecosystems. For example, different manufacturers' smart light bulbs (e.g., Phillips Hue and Sengled) require separate gateway devices for Internet connectivity. Contrast this with Wi-Fi; accessing a new web application from a laptop does *not* require a new Wi-Fi access point.

We implemented *TCPlp*, a full-scale TCP implementation for LLNs [292], and used it to re-examine TCP in LLNs [293] after two decades of evolution. We found that the reasons for poor TCP performance differ from the expected reasons in the literature and developed a set of non-intrusive techniques to make TCP perform well. This has significant implications for IoT system

design, allowing low-power embedded devices to be first-class Internet citizens. *The conclusions of our study have already had impact: they significantly influenced the Thread network standard, developed by a consortium of companies [449] in the IoT space, including Google/Nest, Apple, Qualcomm, and others. TCPlp has been adopted as the TCP stack in OpenThread [360], an open-source implementation of the Thread standard used in some of these companies' products.*

1.4.3 Ghostor [226]

Consider a data-sharing system that allows doctors and their patients to share access to files. An adversary who compromises the storage server can not only learn sensitive information, but also **modify file contents**, causing patients to receive incorrect medical information. Ghostor is a data-sharing system that protects against such an adversary by empowering users to (1) detect server-side integrity violations, and (2) encrypt their files and use the system anonymously. Ghostor achieves (1) by leveraging a blockchain, relying on **decentralized trust**. This is preferable to prior solutions that rely on two **central** servers and assume that at most one is compromised. Alas, the high cost and throughput limitations of blockchain transactions are a serious challenge. Using the blockchain as the second server in an existing two-server design [262] would result in unacceptable overhead, as a blockchain transaction would be required on each data update.

We designed Ghostor to rethink how the system uses the blockchain. In Ghostor, clients interact with the server, not the blockchain, for reading, writing, creating, and sharing files. Periodically, the server posts a cryptographic hash to the blockchain. The hash captures the edit histories of all files, and because the blockchain is a verifiable, append-only ledger, *all clients see the same hash*. Clients check the server's integrity by verifying that the results of their file operations until that point are consistent with the posted hash. We designed Ghostor, like JEDI, to be **inherently flexible.** For example, a Ghostor server might post hashes less frequently, to reduce the cost of blockchain transactions at the expense of less timely integrity verification.

1.4.4 JEDI [296]

JEDI provides end-to-end encryption for communication among IoT devices while preserving the semantics of existing, unencrypted, IoT communication. Two aspects of IoT communication systems make this difficult. First, IoT devices often use publish-subscribe systems that decouple senders from receivers. Second, IoT systems manage fine-grained access control via decentralized delegation—a principal/device with access to resources can delegate access to a subset of those resources to another principal/device for a limited duration.

Policy-based encryption, such as Attribute-Based Encryption (ABE), can support these semantics. Unfortunately, applying ABE-like encryption to all messages is unacceptably energy-intensive for low-power embedded sensors. To address this, we carefully designed JEDI to use coarse-grained timestamps for expiry times (e.g., hour granularity), and to **only require expensive ABE-like encryption/signatures when the timestamp changes** (e.g., once per hour). We also identify a policy-based encryption scheme that is more efficient than ABE yet suitable for JEDI and use it in a non-black-box way, tailored to how JEDI encrypts data. As a result, a sense-and-

send application, sending one reading every 30 seconds on an ultra low-power low-cost Cortex-M0+-based platform, can encrypt its data with JEDI and still achieve several years of battery life. Importantly, our design is **inherently flexible** in its resource demands. For example, on an even more energy-constrained platform, one can use JEDI with coarser-granularity timestamps (e.g., six-hour granularity), to use ABE-like cryptography even more rarely.

1.5 Thesis Statement and Roadmap for This Dissertation

The thesis of this dissertation is that rethinking the way that we design and build systems can help to achieve expressive cryptography's full potential. This should be viewed as an initial version of the thesis statement; Chapter 2 concludes with a refined version of this dissertation's thesis statement based on the background provided in that chapter.

The rest of this dissertation is organized as follows.

- Chapter 2 provides background on expressive cryptography, including what expressive cryptography is and how it can be used. It also provides general observations about expressive cryptography's efficiency and existing techniques for making it more efficient. This knowledge forms the basis for our techniques and explanations later in this dissertation.
- Chapter 3 describes our approach to designing systems for expressive cryptography as a systematization of system design techniques. This systematization of techniques constitutes a framework for understanding and analyzing systems for expressive cryptography. By distilling our system design ideas into this framework, we aim to provide generalizable insights that help others design systems for expressive cryptography.
- Chapter 4, Chapter 5, Chapter 6, and Chapter 7 describe the design and implementation of MAGE, *TCPlp*, Ghostor, and JEDI, respectively. These systems are designed using the techniques in Chapter 3 and provide evidence for the validity and effectiveness of those techniques.
- Chapter 8 discusses related work, with a focus on how the system design techniques in Chapter 3 generalize to existing systems other than the ones described in this dissertation. We use the framework from Chapter 3 to analyze existing systems, pointing out cases where existing systems exemplify our techniques and cases where our techniques could be applied to existing systems. This provides additional evidence for the applicability and effectiveness of our techniques in Chapter 3.
- Chapter 9 describes the impact that our work has had, discusses future research directions, and concludes this dissertation.

Chapter 2

Background

This chapter describes what *expressive cryptography* is and provides background on it. This includes examples of expressive cryptography and how it can be used, observations about its efficiency, and techniques for making expressive cryptography more efficient. Building on that background, this chapter concludes with a precise thesis statement for this dissertation.

2.1 Expressive Cryptography

An important application of cryptography is to control access to information. Public-key cryptography was revolutionary because it allowed one to cryptographically specify which party is allowed to access a message by encrypting the message under that party's public key [83]. In this context, the "expressivity" of a cryptographic scheme refers to expressivity of the control it provides over *who* can access *what* data. We use the term "expressive cryptography" to refer to cryptographic schemes that are more expressive than public-key cryptography.

A universal definition of expressive cryptography, like the one above, is useful but arguably imprecise, partly because not all cryptographic schemes aim to control access to information. For example, some cryptographic schemes (e.g., digital signatures) aim to provide *integrity* guarantees (i.e., guarantees that an adversary has not tampered with data). To clarify this, this section will discuss, through examples, various different types of expressive cryptography.

The term "expressive cryptography" has been used before. For example, Boyen used the term "expressive cryptography" in a 2012 lecture series [82] and a 2013 invited lecture [83]. More recently, Halevi used the term "advanced cryptography" to refer to a similar set of cryptographic schemes [209]. In this dissertation, we prefer the term "expressive cryptography" because "expressive" more accurately captures what makes these schemes interesting from the standpoint of *system design*. Importantly, some cryptographic advances improve security, but not functionality, and we do not consider the resulting schemes to be expressive cryptography. For example, Regev's encryption scheme [395] is *advanced* in that it provides protection against quantum adversaries, but *not expressive* because it is semantically identical to standard public-key encryption. Using a quantum-secure public-key encryption scheme in a system instead of regular public-key encryp-

tion can make the system secure against a quantum adversary (i.e., it can strengthen the security guarantees) but it cannot enable new functionality under the existing threat model. In contrast, expressive cryptography can enable, under the same threat model, support for richer applications (i.e., more functionality) compared to standard public-key cryptography. Where applicable, our descriptions below include explanations of how each type of expressive cryptography enables new kinds of systems.

2.1.1 Expressive Cryptography for Protecting Confidentiality

Boyen describes expressivity as the ability to "express, in increasingly powerful and flexible ways, the exact beneficiaries of the right to decrypt a given ciphertext" [83]. This description applies well to encryption schemes, which aim to protect the confidentiality of encrypted data. We refer to these expressive encryption schemes, collectively, as *policy-based encryption* to reflect the fact that the recipients of a message may be specified as a policy rather than as a public-key known at the time of encryption. Below, we will see how expressive cryptography allows the encryptor to more flexibly describe who can decrypt a ciphertext.

2.1.1.1 Identity-Based Encryption

An identity-based encryption (IBE) scheme [422, 65] works as follows. When encrypting a message, a party identifies the intended recipient as an *arbitrary string of bytes* called an *ID*. A party is given the secret key for her ID, which allows her to decrypt any message encrypted for her ID. This is orchestrated by a party called the authority who creates an *IBE system*, which consists of public parameters (sometimes called a master public key) and a master secret key. The authority can use the master secret key to generate a secret key for any ID and give it to the party with that ID.

An important difference between identity-based encryption and public-key cryptography is that IBE allows the encryptor to specify the intended recipient as a string of bytes rather than as a public key. This makes IBE more flexible than public-key cryptography in two ways. First, it allows the encryptor to specify the recipient of the message without having to look up the recipient's public key; once a party obtains the public parameters for an IBE system, she can encrypt messages for any ID in that system. Second, it allows the encryptor to encrypt a message for an ID *even if the secret key for that ID has not yet been created*. In contrast, public-key encryption only allows encrypting a message for a recipient after the recipient has generated her keypair and the encryptor has learned the recipient's public key.

2.1.1.2 Attribute-Based Encryption

Attribute-based encryption (ABE) [203] has a similar setup to IBE, but is more flexible, in the following sense. When encrypting a message, a party chooses *multiple* arbitrary strings of bytes called *attributes*. Secret keys no longer correspond to a single ID; they correspond to *access policies* represented as logical expressions checking for the presence of certain attributes. A secret

key can decrypt a ciphertext if the attributes associated with the ciphertext satisfy the access policy associated with the secret key.

The type of ABE described above is sometimes called *key-policy ABE*, or KP-ABE, to reflect the fact that secret keys are associated with policies and ciphertexts are associated with attributes. In another type of ABE, called *ciphertext-policy ABE* [51], or CP-ABE, secret keys are associated with attributes and ciphertexts are associated with policies. In CP-ABE, a secret key can decrypt a ciphertext if the attributes associated with the secret key satisfy the access policy associated with the ciphertext.

IBE can be understood as a special case of KP-ABE where only one attribute can be used per ciphertext and policies are restricted to checking for the presence of a single attribute. In that sense, ABE subsumes IBE and is strictly more expressive. To intuitively understand how ABE enables new functionality from the standpoint of system design, consider an application where multiple parties must be given access to a single message. With public-key encryption or IBE, the encryptor would have to enumerate the IDs or public keys of all recipients when encrypting a message. In contrast, CP-ABE allows the encryptor to specify a *policy* of who is allowed to access it, without having to know exactly who the recipients are. If a new user joins the system, they will automatically be able to decrypt the ciphertext if their combination of attributes matches the policy associated with the ciphertext, even if no previous user had that particular combination of attributes.

Over the years, increasingly expressive variants of ABE have been developed. Schemes like Hierarchical IBE (HIBE) [224, 186, 63] and Fuzzy IBE [404] predate the term "attribute-based encryption," but they can be considered primitive ABE schemes that only support a limited set of policies—checking for the presence of attributes restricted to a hierarchical structure and checking that the number of matching attributes is above a certain threshold, respectively. The first schemes that were called ABE [203, 51], allowed monotonic access policies—informally, policies that check for the presence, but not the absence, of certain attributes. Newer ABE schemes support more general access policies including support for non-monotonic formulas [363] and Boolean circuits [200].

Multiple system design proposals in the context of cloud computing leverage ABE [468, 501]. In the context of cloud computing, ABE allows users to bake access control policies directly into ciphertexts of their data and enforce those policies cryptographically. This gives users control of access policies governing their data, without placing users at the mercy of the cloud provider to respect and enforce those access policies. Another compelling application of ABE is in the Internet of Things (IoT) space [355, 197]. Cryptographically enforced access control is compelling in the IoT space because IoT devices typically do not have trusted storage and the complex access policies supported by ABE are attractive in the context of IoT applications.

2.1.1.3 Predicate Encryption

While ABE can allow complex policies to be attached to keys or ciphertexts, ABE only hides the message contents, *not the policies or attributes attached to a ciphertext*. Predicate encryption schemes can be thought of as an extension of ABE where the policies and attributes attached to

a ciphertext are also cryptographically hidden. To emphasize this distinction, some of the literature refers to ABE as *public-index predicate encryption* or ABE's security guarantee as *payload hiding*, and to schemes that hide the policies and attributes attached to a ciphertext as *private-index predicate encryption* or the corresponding security guarantee as *attribute hiding* [265, 68]. This dissertation uses the term "predicate encryption" to refer to private-index, attribute-hiding schemes.

As with ABE, increasingly powerful predicate encryption schemes have been developed over time. Anonymous IBE [64, 1] and anonymous HIBE [84] predate the term "predicate encryption," but they can be considered primitive predicate encryption schemes that support the simple policies of IBE and HIBE—checking equality on a single attribute and checking for the presence of attributes restricted to a hierarchical structure, respectively. In the following years, more sophisticated predicate encryption schemes, like Hidden Vector Encryption [70] and Inner Product Encryption [265], emerged. These schemes support more sophisticated policies, including conjunctions and disjunctions over attributes, but cannot support arbitrary combinations of conjunctions and disjunctions over many attributes as efficiently as classical ABE schemes [203]. Later, the community developed more advanced schemes [403, 201], including Worry-Free Encryption, that support more general policies. These schemes additionally generalize predicate encryption into *functional encryption* [68], in which policies do not merely dictate whether a message can be decrypted, but *transform* the message obtained by decryption.

2.1.2 Expressive Cryptography for Protecting Integrity

A number of cryptographic techniques, including digital signatures, aim to protect message integrity. In the context of such techniques, "expressivity" refers to increasingly powerful and flexible guarantees about data being delivered correctly. That said, expressivity can also be used in other ways, to refer to the other information revealed by the scheme. For example, group signatures [106] are more expressive than standard digital signatures in the sense that the signer of a message remains anonymous within a group of signers, even to parties who can verify her signature.

2.1.2.1 Transparency Logs

Transparency logs, sometimes referred to as verifiable data structures, are public, authenticated data structures that provide integrity guarantees for the operations performed on a data structure. For example, Certificate Transparency [305] is a log data structure that guarantees that data will only be appended, never removed, and that all users see a consistent view of the log. As another example, Key Transparency [342] is a map data structure that guarantees that the results of multiple users' queries are consistent with one another. Transparency logs often derive their cryptographic security guarantees from on cryptographic data structures like Merkle trees.

To understand the value of transparency logs, consider the problem of HTTPS certificates. Certificate Transparency is designed to be used with HTTPS certificates; clients only use an HTTPS certificate if they can find it in a trusted Certificate Transparency log. The data structure guarantees

that the organizer of the web domain sees the same certificates as the user, assuring users that a fraudulent certificate would be promptly detected and dealt with.

2.1.2.2 Blockchains

A blockchain [282] is an append-only log consisting of items called *transactions*. They are designed to only accept transactions into the log that are valid, in a sense particular to the blockchain. For example, a blockchain for a cryptocurrency, like Bitcoin, may only accept transactions in which a user spends a token (i.e., "Bitcoin") still in her possession. Blockchains achieve this by relying on users (i.e., "miners") to decide, in a decentralized manner, on which transactions to include. This procedure is similar to a probabilistic vote of sorts; to protect against Sybil attacks, users' votes are weighted based on their computational power (i.e., "proof of work") or wealth in tokens (i.e., "proof of stake").

At a high level, blockchains are similar to transparency logs. Each is a data structure together with a protocol that make guarantees about the operations that are performed on the data structure. However, there are important differences. First, while transparency logs are hosted by a single, centralized party, blockchains are fully decentralized. Second, while transparency logs guarantee merely that users can detect if an invalid operation is performed, blockchains can deny invalid transactions outright. One consequence of this is that, while a transparency log's availability depends on the party hosting the log, blockchains do not require trust in a central party for availability. As a result of these differences, blockchains may be considered more expressive than transparency logs.

The original application of blockchains was cryptocurrency [353]. Due to their ability to provide consensus and consistency in an open-membership setting, blockchains have become a system design primitive in their own right and have evolved into a substrate for decentralized applications [93]. For example, there have been proposals to leverage blockchains to build fully decentralized systems for data storage [442] and virtual reality [361]. Closed-membership versions of blockchains, sometimes called "private blockchains," have also been developed. Private blockchains give up true decentralization in exchange for other useful properties like efficiency, privacy, and central oversight. An important application of private blockchains is to improve transparency and auditability in business supply chains [138].

2.1.3 Expressive Cryptography for Protecting Computation

Some cryptographic techniques not only protect data, but also incorporate computation in some way. In the context of expressive cryptography for protecting computation, an important measure of "expressivity" is the generality and flexibility of computation that can be protected. We focus in this section on tools that allow for computing on encrypted data. Predicate encryption and functional encryption are listed in Section 2.1.1 due to their relationship to ABE, but they can also be considered as protecting computation and could be listed in this section. Other cryptographic techniques like Zero-Knowledge Proofs [194] and Oblivious RAM [193] do not allow for computation on encrypted data but provide other cryptographic guarantees relating to computation. They are

also forms of expressive cryptography, but we do not focus on them in this section because they are not directly related to the work in this dissertation.

2.1.3.1 Secure Multi-Party Computation

Secure Multi-Party Computation (SMPC) [496, 192, 46] allows n parties, with secret inputs $x_1, \ldots, x_n$ (where party i has secret input x_i), to work together to compute an arbitrary function f of their choice on their secret inputs. SMPC allows the parties to learn $f(x_1, \ldots, x_n)$, but guarantees that they will learn nothing else about the other parties' secret inputs.

Early SMPC protocols, such as Yao's Garbled Circuits [496], GMW [192], and BGW [46] could support functions representing any fixed-size computation. They achieved this by representing f as a logical circuit—for example, f may be a circuit where wires contain bits and gates represent boolean operations, or f may be a circuit where wires contain integers in a Galois Field and gates represent addition or multiplication operations.

Later schemes built on these in several ways. Some later schemes improve expressivity by supporting SMPC for more parties. For example, while Yao's Garbled Circuits protocol originally supported only two parties ($n = 2$), the BMR protocol [40] supports SMPC for more than two parties ($n > 2$) using a generalization of Yao's Garbled Circuits that preserves its round complexity. Others improve security but not expressivity, supporting stronger threat models. For example, although the original Yao's Garbled Circuits construction is secure only against a semi-honest adversary, it can be augmented with cut-and-choose techniques to be secure against a malicious adversary [319]. Finally, some actually reduce expressivity by supporting only a few specific kinds of functions f, rather than general functions. For example, specialized SMPC protocols exist for set intersection [178, 377] and digital signature generation [428], even though generic SMPC protocols, like Yao's Garbled Circuits, are capable of the same functionality [229]. The advantage of specialized SMPC protocols is that they may be more performant for the functions that they support than generic SMPC protocols, as we discuss later in Section 2.2.2.

A compelling application of SMPC is to enable *secure collaborative data analytics*. In many cases, data analytics jobs require data that reside across multiple organizations, yet organizations cannot directly share data due to legal regulations and privacy concerns. We discussed secure collaborative data analytics applications in Section 1.1 to motivate our work—such applications include collaboration among hospitals for research, collaboration among financial institutions to detect fraud, and measurement of social issues like the gender wage gap. SMPC provides a solution to such problems, allowing n organizations to compute a data analysis of their choice, described by a function f, over their private datasets $x_1, \ldots, x_n$. Research proposals like SMCQL [38] and Conclave [462] are systems based on SMPC that directly target secure collaborative data analytics.

Secure collaborative data analytics has seen real-world adoption and significant industry interest [196]. As we explained in Section 1.1, Google is exploring using SMPC to compute ad conversions—how many users who viewed an ad become customers [464, 243]. SMPC is needed because Google does not want to reveal the viewership of an ad, and the company running ads on Google does not want to reveal who their customers are; SMPC allows them to compute ad conversions without revealing additional information. Two of Meta's products, Private Lift and

Private Attribution, are based on SMPC and are actively used by advertisers [396]. These products solve a similar problem to the "ad conversions" problem addressed by Google—they allow users to perform randomized controlled trials to assess the impact of advertising on the conversion rate. Another example of real-world usage of SMPC, which we explained in Section 1.1, is the deployment of SMPC in the Boston area to measure social issues like the gender wage gap and economic inclusion of minority-owned businesses [386].

SMPC also enables *distributed trust*. A class of systems aims to avoid trusting a single server by splitting trust across n servers (for $n > 1$) [126, 111]. For example, instead of a user storing her signing key (or some other secret) on a single device, she may instead store secret shares of her signing key on n devices. In order to use her signing key, the user would have those n devices run an SMPC protocol to produce the desired signature. This approach is used in the cryptocurrency space; companies like Fireblocks provide SMPC-based wallets used to secure millions of dollars in digital assets [172, 173]. Researchers have also used distributed trust based on SMPC to enable more complex systems than digital signature generation. For example, researchers have designed SMPC-based distributed trust systems for collection of aggregate statistics [127] and for data storage and sharing [112].

2.1.3.2 Homomorphic Encryption

Homomorphic encryption allows a party who holds $\mathsf{Enc}(x)$ to directly compute $\mathsf{Enc}(f(x))$, for certain types of functions f, without first decrypting $\mathsf{Enc}(x)$. Multiple ciphertexts encrypted with the same key can also be combined; a party who holds $\mathsf{Enc}(x)$ and $\mathsf{Enc}(y)$ can compute $\mathsf{Enc}(f(x,y))$. Importantly, the party who computes this does not need to know the secret key to decrypt the ciphertext and never sees x, y, or $f(x,y)$ in plaintext—the data remain encrypted throughout the computation.

Early homomorphic encryption schemes supported only simple types of functions. For example, the ElGamal encryption scheme [162] supports multiplication and the Paillier encryption scheme [367] supports addition. A more expressive encryption scheme, developed by Boneh et al. [67], supports logical circuits of additions and multiplications with a multiplicative depth of 1—enough to support functions described by 2-DNF formulas, but not general functions.

A breakthrough was Gentry's development of Fully Homomorphic Encryption (FHE) [182], a type of homomorphic encryption that supports general functions. As Gentry's scheme supports general functions, later works on homomorphic encryption do not aim to improve the generality of the computation—they aim to simplify the design, improve performance, or improve expressivity in ways other than the generality of computation (e.g., creating an FHE scheme that supports policies like those in ABE [185]). Performance has been a particular focus of followup work [86, 148, 116, 115]. Some of these works alter the computation model; for example, *leveled* FHE schemes like BGV can efficiently evaluate add-multiply circuits but require the depth to be relatively small and known at setup time [86], and FHE schemes like CKKS support approximate arithmetic [115].

Homomorphic encryption has been used in systems designed to compute on encrypted data, usually to support computing aggregates such as the sum of a dataset [381, 420, 369, 419]. To compute a particular aggregate, simple *Partially Homomorphic Encryption* schemes that support

a simple type of function, such as the Paillier encryption scheme (or a symmetric-key equivalent), are typically sufficient. For more complex computations over encrypted data, such as machine learning inference, leveled FHE schemes are sometimes useful [79]. Homomorphic encryption is also a useful building block as part of other cryptographic protocols. For example, specialized SMPC protocols, such as those used for neural network inference, sometimes use homomorphic encryption for part of the computation [260].

2.2 Efficiency and Overheads of Expressive Cryptography

Cryptographic operations like encryption can require a significant amount of computing resources. We use the term "cryptographic overhead," or simply "overhead," to refer to the extra resources required due to cryptography. This section presents more details on the overhead of using expressive cryptography. As we will see, the overhead of expressive cryptography is, in many cases, a significant obstacle to applying it in systems.

2.2.1 Types of Cryptographic Overhead

Systems developers often think of cryptographic overhead in terms of computation time—for example, the CPU time required to encrypt or decrypt messages. Expressive cryptography, however, is expensive not only in terms of CPU time, but also in terms of other computing resources like memory and networking. This section explores the ways in which expressive cryptography can be expensive.

2.2.1.1 CPU Overhead

Expressive cryptography consumes a significant amount of CPU time, typically more than regular, non-expressive cryptography. For example, ABE encryption *with a single attribute* can require two orders of magnitude more CPU time than encryption with public-key cryptography [477]. Furthermore, the CPU time for ABE encryption increases linearly in the number of attributes. Other expressive encryption schemes, like predicate encryption and fully homomorphic encryption, also have high overheads for encryption.

For expressive cryptography that supports computing on encrypted data, there is also a significant overhead to running a computation on encrypted data compared to running the same computation on plaintext data. FHE and SMPC primitives that support generic computation can require orders of magnitude more CPU time than computing directly on plaintext. To get a sense of how much slower, consider JustGarble [45], a system for evaluating Yao's Garbled Circuits that is optimized for efficiency. JustGarble can evaluate a gate in 7.25 ns, amortized, based on a circuit consisting of 82% XOR gates (where XOR gates are particularly efficient to execute). Based on this measurement, adding two 32-bit integers would take about 1 microsecond (assuming 4 XOR gates and 1 AND gate per bit of the addition). This is about three orders of magnitude slower than adding two 32-bit integers directly in the processor. Computation over encrypted data in FHE

schemes is also much slower than computation over plaintext data. For example, TFHE [116] requires about 10 milliseconds to execute a binary gate, which is about $10,000\times$ slower than JustGarble and therefore about seven orders of magnitude slower than computing directly in the processor. Leveled FHE schemes can be faster than TFHE, as they can operate on batches of data in a SIMD fashion and can perform integer operations directly instead of just binary operations. In these schemes, addition can be very fast, but multiplication can still take about 10 milliseconds (rounded to order of magnitude) [459]. When accounting for SIMD-style batches of approximately 1,000 to 10,000 elements, multiplication in these schemes may be approximately three to four orders of magnitude slower than operating on plaintext. While the above overheads apply to the computation itself, there can be additional overhead associated with transforming the desired function to a logical circuit representation suitable for cryptographic computation.

Blockchains are a special case. Miners in a proof-of-work-based blockchain incur significant CPU overhead due to the proof of work mechanism for mining blocks. Yet a faster CPU would not make mining faster, as the difficulty of the proof-of-work problems is adjusted to keep the time between blocks fairly constant. It would merely increase the profitability of mining for miners with the faster CPU, enabling those miners to more effectively compete with other miners.

2.2.1.2 Memory Overhead

Expressive cryptography can have high memory overhead because ciphertexts are large. For example, when using 2048-bit RSA (standard, non-expressive public-key cryptography), a ciphertext is 256 bytes. Elliptic-curve-based ciphertexts (e.g., with ElGamal encryption) can be even smaller, just tens of bytes. In contrast, an ABE ciphertext can be *kilobytes* in size. While a few kilobytes per ciphertext may not seem large in an absolute sense, it can be very significant for resource-constrained embedded devices. For example, ultra low-power wireless sensing platforms may have only several tens of kilobytes of RAM, so encrypting sensed data points with expressive cryptography may be a significant memory burden.

For schemes like IBE and ABE, this matters most for small messages, which become much larger when encrypted. This is because, for large messages, one can use hybrid encryption. With hybrid encryption, one randomly samples a symmetric key k, uses IBE or ABE (or standard public-key encryption) to encrypt k, and then uses k to encrypt the actual message. Thus, the size of the ciphertext can be amortized by the size of the message itself, for large messages.

Hybrid encryption is not suitable, however, for expressive cryptography that supports computation on encrypted data. This means that ciphertext expansion applies to all inputs and intermediate results when performing the desired computation on encrypted data. For example, in Yao's Garbled Circuits protocol [496], each wire in the circuit represents one bit of plaintext but corresponds to a 16-byte label during execution of the protocol. This means that evaluating the garbled circuit for a function f will require $128\times$ more memory than evaluating f in plaintext. Secret-sharing-based SMPC protocols [192] do not have an inherent expansion factor, since secret shares are, in principle, the same size as plaintext data. Integer-based secret sharing schemes [46], however, still have an expansion factor because all data items, even bits or small integers, must be represented as full-size secret shares. FHE schemes also have a large expansion factor. In TFHE [116], for

example, a ciphertext represents a single bit but may be multiple kilobytes in size. This represents a memory overhead of $10,000\times$ or more compared to plaintext computation. Leveled FHE schemes (e.g., CKKS [115]) may have ciphertexts that are tens or hundreds of kilobytes. This may be amortized by using batching, but memory overheads may still be an order of magnitude or more if the application at hand does not allow using batching to its fullest.

2.2.1.3 Networking Overhead

Network overhead can manifest in two forms: bandwidth and latency. We explore each below.

Systems based on expressive cryptography often transfer ciphertexts over the network. For example, a resource-constrained device might encrypt data with FHE and send the ciphertexts to a more powerful computer to offload computation to it, or it might encrypt data with ABE to allow rich, cryptographically-enforced access control policies and send the ciphertexts to a database for later retrieval by users. For such systems, large ciphertext sizes not only cause memory overhead (as described in Section 2.2.1.2), but can also result in network bandwidth overheads when transferring ciphertexts over the network. For example, just as an ABE ciphertext that is kilobytes in size may present significant memory overhead for an ultra low-power embedded sensing device, it may also present significant networking overhead for such a device. In particular, such devices may use low-power wireless networks, which have limitations, such as low bandwidth, that make it difficult to perform bulk data transfer. Even for server-class devices with plenty of provisioned network bandwidth, a bulk transfer of many ciphertexts may present significant network bandwidth overhead.

SMPC protocols inherently require multiple parties and can consume significant amounts of network bandwidth among these parties. For example, we saw that JustGarble can evaluate Yao's Garbled Circuits while spending just 7.25 ns per gate, on average, in a workload where $\approx 80\%$ of the gates are AND gates. With the half gates optimization [504], each AND gates requires transferring 32 B between the two parties. Thus, Yao's Garbled Circuits requires about 8 Gbit/s of network bandwidth to fully utilize a *single CPU core* of the party evaluating a garbled circuit.

Other types of SMPC protocols add overhead in the form of network latency. For example, secret-sharing-based SMPC protocols require a round of communication per nonlinear operation (e.g., AND of bits or multiplication of integers). Because each round of communication involves a network round-trip between pairs of parties, these operations may incur significant overhead in the form of network latency, particularly when the network round-trip time between parties is large. The overhead can be particularly high when evaluating functions that, when represented in circuit form for SMPC, have a high depth in nonlinear operations.

Transparency logs and blockchains also incur network overhead. Participants in a blockchain incur network overhead to discover newly mined blocks. Systems like Certificate Transparency require users to "gossip" to ensure that the log server presents the same view of the log to all users. Other transparency log schemes (e.g., key transparency) require users to monitor the log to ensure that its state is consistent with their expectations (leveraging the fact that the log guarantees that all users' views of the log will be consistent).

2.2.2 Trade-Off Between Expressivity and Efficiency

As a general rule, *more expressive cryptographic schemes tend to be less efficient*. Stated differently, when choosing a cryptographic scheme, there is usually a trade-off between expressivity and efficiency.

For example, consider expressive cryptographic schemes that protect confidentiality. KP-ABE is strictly more expressive than HIBE, and HIBE is strictly more expensive than IBE. Accordingly, KP-ABE requires more CPU time and has larger ciphertexts than HIBE, and HIBE requires more CPU time and has larger ciphertexts than IBE.

As discussed in Section 2.1.2, blockchains may be considered more expressive than transparency logs due to their fully decentralized nature and ability to outright deny invalid transactions. This comes, however, at significant performance costs—blockchains like Bitcoin have low transaction throughput and high transaction latency (e.g., tens of minutes to an hour for a transaction to be accepted in the blockchain).

The same applies to cryptography for computing on encrypted data. Generic SMPC protocols can execute any function f, but it may be possible to compute f more efficiently using a specialized SMPC protocol for that task. For example, the specialized SMPC protocols for set intersection are more efficient, in some settings, than using generic SMPC protocols to compute those functions [377]. Similarly, homomorphic encryption schemes that support only simple types of functions can be more lightweight than FHE.

2.3 Techniques for Making Expressive Cryptography Efficient

Given the enormous potential of expressive cryptography, there has been much effort to make the cryptography more efficient. There have been three main approaches to doing so. The first approach is to create *generic theoretical improvements* that improve the efficiency of generic schemes. The second approach is to create *specialized cryptographic schemes* that improve the performance of expressive cryptography by specializing it to the particular application at hand. The third approach is to apply *systems techniques* to make expressive cryptography more efficient. We describe these approaches below.

2.3.1 Generic Theoretical Improvements

An important approach to making an expressive cryptographic scheme efficient is to improve the underlying mathematics to make the scheme more efficient. This approach is particularly powerful because any system that uses the relevant cryptographic scheme will benefit from the resulting improvements.

Yao's Garbled Circuits protocol [496] is one example of expressive cryptography that has benefited from this. Generic theoretical improvements, such as Point-and-Permute [40], Free XOR [284, 283], and Half Gates [504] have significantly improved the computation, memory, and network bandwidth required by the protocol. Oblivious Transfer [387], the first step in using garbled circuits for SMPC, has also seen generic theoretical improvements [244, 85, 493]. Because

these improvements did not change the syntax (i.e., interface) of the cryptographic scheme, they fall under the category of "generic improvements."

Sometimes, it is necessary to make small changes to the interface in order to obtain performance gains. The resulting improvements can still be considered "generic" if they do not significantly diminish the space of applications that can benefit from them. For example, Gentry's initial FHE scheme [182] was followed by leveled FHE schemes [86, 115]. These leveled FHE schemes changed the syntax of the schemes to require the maximum multiplicative depth to be known at setup time, but this minor syntactic change resulted in significant performance improvements, since applications whose multiplicative depth is relatively shallow and known in advance can avoid bootstrapping. As another example, early HIBE schemes [186] were followed by the BBG HIBE scheme [63], which requires the maximum hierarchical depth to be known at setup time (as in certain prior works [62]) but allows for smaller ciphertexts than prior schemes.

2.3.2 Specialized Cryptographic Schemes

An alternative to developing generic improvements is to leverage the "expressivity vs. efficiency" trade-off Section 2.2.2 to trade generality for efficiency. Simply put, we can specialize the cryptographic scheme to the application at hand in order to improve its performance.

One clear example of this is the development of SMPC for specialized functions. For example, specialized Private Set Intersection (PSI) protocols implement SMPC for a particular function that takes as inputs a set of values from each party and outputs the intersection of those sets. While it is possible to implement PSI by evaluating a function f for set intersection using a generic SMPC protocol like garbled circuits [229], state-of-the-art specialized PSI protocols [377, 56] outperform such approaches based on generic SMPC. As a result, designer who needs to run a workload based on Private Set Intersection (PSI) would, in many cases, be better off leveraging a specialized PSI protocol than instantiating PSI with a generic SMPC protocol.

One can specialize cryptographic protocols not only for useful functions, but also directly for applications. The research community has developed specialized SMPC protocols for data analytics [378] and for machine learning algorithms like k-means [251], linear models [358, 515], and neural networks [346, 260, 291, 392, 306, 356]. Some of these efforts not only specialize the cryptography, but also make changes to the model to admit more efficient cryptographic constructions [397, 345].

2.3.3 Systems Techniques

One can also improve the performance of expressive cryptography by building systems to better support it. This approach can bring efficiency improvements without the need for theoretical advances or changes to the underlying mathematics of expressive cryptography. A classical approach to making cryptography faster in this way is to manually write assembly code to optimize the relevant procedures. While this remains a useful approach, expressive cryptography necessitates a more holistic approach, as its overheads manifest not only in CPU time, but also in memory and network usage.

One example of how systems techniques can improve the performance of expressive cryptography is the development of execution frameworks for Yao's Garbled Circuits. Early systems for executing garbled circuits, like Fairplay [332], required a large amount of memory because they would materialize the entire garbled circuit in memory at each party. Later, the HEKM system [230] improved memory consumption significantly by avoiding materializing the entire garbled circuit at either party. This reduced the memory overhead significantly, as each party only had to allocate memory for all of the wires in the circuit, not all of the garbled gates. Subsequent garbled circuit frameworks [288, 287] improved on HEKM further, allocating memory for just the live wires rather than all wires in the circuit. TinyGarble [436] applied logic synthesis techniques to optimize the boolean circuit representation of the function f to execute. These systems improvements significantly improved the performance of garbled circuits, at times producing comparable improvements to developing specialized cryptographic protocols [229].

In concurrent work, researchers applied systems to help navigate the performance trade-offs among cryptographic protocols. Tools like EzPC [103], Cerebro [514], and Silph [108] help to automate the design of specialized cryptographic protocols for applications like machine learning. This is enabled by cryptographic work like ABY [139], which provides flexible ways to combine different SMPC techniques with different performance trade-offs, and systems work like CostCo [165], which provides cost models for SMPC protocols that these tools can use to identify the most efficient designs. We discuss these techniques in greater depth in Chapter 8.

Concurrently with the work presented in this dissertation, systems techniques have also been applied to help applications use expressive cryptography in an efficient way. For example, DeepSecure [399] improves the performance of deep learning inference with Yao's Garbled Circuits by performing preprocessing outside of SMPC. SMCQL [38] and Conclave [462] improve the performance of data analytics in SMPC in a similar way, by identifying portions of the computation that can happen in a preprocessing phase, to minimize the computation that must be protected cryptographically with SMPC. These works leverage some of the techniques that this dissertation develops, so we discuss them in greater depth in Chapter 8.

Another line of work concurrent with this dissertation aims to make expressive cryptography faster by leveraging hardware accelerators. For SMPC protocols for neural network inference and training, researchers have designed cryptographic protocols that can be accelerated using GPUs [445] and designed the necessary systems support to effectively leverage GPUs to accelerate these protocols end-to-end [479]. For FHE protocols, researchers developed F1 [168] and CraterLake [406], hardware accelerators that can provide orders of magnitude of speedup for FHE workloads compared to relying only on the CPU.

2.4 Conclusion and Thesis Statement Revisited

Expressive cryptography provides useful functionality for many real applications and has enormous potential. SMPC, for example, is seeing significant industry interest, with deployment by Google and Meta and the potential to solve important, real-world problems. Unfortunately, expres-

sive cryptography can be expensive—it consumes lots of computing resources. This is a significant obstacle to realizing expressive cryptography's full potential.

A natural question is how we can overcome the efficiency-related challenges of expressive cryptography and allow it to fulfill its enormous potential. One approach where we have seen significant progress, and where we will continue to see progress, is in making theoretical advances. Such approaches include generic improvements to the underlying mathematics to make cryptographic schemes inherently faster and specializations of cryptographic schemes to particular functionalities or applications to trade off generality for efficiency. The theory has advanced so far, however, that important applications of expressive cryptography are within striking distance of practicality. In many cases, performance bottlenecks emerged not because the cryptography was inherently slow, but because the systems built around expressive cryptography had not kept up with the theoretical advances. This has led to advances in systems relating to expressive cryptography, both for systems that use expressive cryptography and for systems that provide frameworks for executing expressive cryptography. As an example, frameworks for executing garbled circuits have improved tremendously, with state-of-the-art garbled circuit frameworks capable of executing hundreds of millions of boolean gates per second on a single CPU core. SMPC schemes for particular functionalities and applications are even more efficient. Many of the advances in system design were developed concurrently with the work in this dissertation.

The thesis of this dissertation, refined from our initial statement in Chapter 1, is that rethinking the way that we design and build systems can help make expressive cryptography less expensive to use, making expressive cryptography practical for more applications and thereby helping to achieve expressive cryptography's full potential. While we are not the first to explore how systems should use expressive cryptography or how system design might make expressive cryptography more efficient, we systematically study the interplay between expressive cryptography and system design to provide practical insights and guidance on how to design systems for expressive cryptography. The results are two-fold. First, we develop new system design techniques and explore them through the design, implementation, and evaluation of four new systems: MAGE, *TCPlp*, Ghostor, and JEDI. Second, our insights and guidance provide a framework for understanding and analyzing systems for expressive cryptography. This framework can, of course, be applied to the four systems presented in this dissertation. But it can also be used to analyze systems for expressive cryptography developed independently of this dissertation, as we do in Chapter 8.

Chapter 3

System Design Techniques for Expressive Cryptography

This chapter describes system design techniques for expressive cryptography. In doing so, we provide a framework to understand, analyze, and design systems for expressive cryptography. We identify two distinct high-level approaches that one could take to rethink system design for expressive cryptography. First, one can rethink how to design the underlying systems that *support* expressive cryptography to do so efficiently. Second, one can rethink how applications, and the systems that directly support those applications, should *use* expressive cryptography to bring security guarantees to users. We present system design techniques for expressive cryptography divided into two classes (Section 3.1 and Section 3.2), one for each of these two approaches. The chapter concludes with a discussion of when to apply each class of techniques when designing a system.

We explain the techniques primarily using examples from the work in this dissertation, in effect applying our framework to our own work. That said, we do not claim that the system design techniques presented in this section are novel. Some of these techniques have been used before in work prior to and concurrent with ours; we include pointers to other systems where they are particularly relevant. Our framework can also be applied to analyze systems relating to expressive cryptography developed independently of this dissertation, as we explore in Chapter 8.

3.1 Designing Systems that Support Expressive Cryptography

The first class of techniques applies to designing the underlying systems on which expressive cryptography depends. An example of such a system is the operating system (e.g., Linux), which provides the process abstraction inside which expressive cryptography runs and which gives expressive cryptography access to hardware resources like CPUs, memory, and the network. This section focuses on techniques for designing the underlying system in such a way that expressive cryptography becomes more efficient. In some cases, the techniques presented here benefit applications beyond just expressive cryptography, and therefore may be of broader interest to the systems research community. In our explanations below, we discuss when this is the case.

3.1.1 Manage Resources According to the Structure of the Computation

The first technique is to manage resources according to the structure of the computation. Whereas standard, off-the-shelf systems manage resources in order to support *general* programs, expressive cryptography may use resources in a *special* way that admits more efficient forms of resource management. By designing the underlying system to leverage the structure of cryptographic computation, one can significantly improve resource management, which can in turn make expressive cryptography more efficient to use.

We applied this technique in MAGE to make it more efficient to compute on encrypted data. Schemes for computing on encrypted data can have very high memory overhead. This can be a significant obstacle for using these schemes in applications—if the application does not fit in memory, then the operating system starts paging data to a storage device (e.g., an SSD), which makes the application run much more slowly. Applications like secure collaborative data analytics are particularly problematic because they compute on large amounts of data, which expand multiplicatively when applying the relevant cryptographic schemes. Our key observation in MAGE is that, although FHE and generic SMPC schemes support general computation, they require the computation f over encrypted data to be structured in a particular way that admits a fundamentally more efficient form of memory management. Specifically, they require f to be expressed as a logical circuit, where each gate corresponds to an operation on encrypted data directly supported by the cryptographic scheme. As a result, the sequence of memory accesses in evaluating f is (1) deterministic, and (2) independent of the values of the inputs to f. This allows MAGE to analyze the code for the function f and precompute the sequence of memory accesses that will be issued when evaluating f. Thus, MAGE can manage memory with foreknowledge of f's future memory accesses, allowing it to make more accurate and timely decisions about which pages to prefetch from storage into memory and which pages in memory to evict to storage.

A key property that MAGE relies on—that the memory access pattern of f is independent of f's inputs—is called *obliviousness*. Obliviousness is inherent to the security guarantees of computing on encrypted data. If a scheme for computing on encrypted data were to work with non-oblivious programs, then the party executing the program over encrypted data could infer something about the inputs to the program from the memory accesses, undermining the security of the cryptographic scheme. Thus, we expect MAGE's techniques to generalize beyond the specific cryptographic schemes that we explore in our work. In fact, the techniques may generalize to workloads beyond just cryptographic schemes, as certain plaintext workloads, such as matrix multiplication and neural network inference, are also oblivious and deterministic. We discuss this further in Section 9.2.1.

Another example of this technique is to write assembly code by hand to optimize a particular cryptographic routine. It is often possible to do better than the underlying system (the compiler, in this case) due to knowledge of the structure of the relevant cryptographic routines, even though the compiler, in principle, could obtain this knowledge from analyzing the source code. For example, we used this technique in JEDI to optimize the bilinear group implementation underlying WKD-IBE. By writing assembly code for big integer arithmetic specialized to the relevant integer sizes, we significantly improved performance compared to compiler-generated assembly code.

For example, our implementation optimizes the number of data transfers between registers and the cache, which contributes to our overall performance improvement.

MAGE demonstrates that even generic cryptographic schemes may have particular structures that admit specialized resource management strategies in the underlying system. Specialized cryptographic schemes may admit more opportunities for specialized resource management.

3.1.2 Identify the Bottleneck and Generically Optimize It

The second technique is to identify the bottleneck and generically optimize it. Sometimes, cryptographic applications place additional strain on certain resources. As a result, the underlying system's resource management is burdened in ways that do not arise for non-cryptographic systems. Any improvement to the underlying system that improves resource management for strained resources, therefore, can benefit cryptographic systems. As such improvements are not necessarily cryptography-specific, they often broadly benefit applications beyond those that use expressive cryptography.

For example, network overheads can be significant for expressive cryptography. Part of this is due to larger ciphertext sizes—a number of proposals for IoT security involve using expressive encryptions schemes like ABE, and these proposals often involve sending ciphertexts over the network. Due to the expansion factor of expressive encryption schemes, the network overheads can be significant. *TCPlp* addresses this by providing a means for reliable, high-throughput data transfer over low-power wireless networks, making it easier to transfer large ciphertexts. Network overheads are significant for other kinds of expressive cryptography as well, including SMPC. Techniques that improve network performance in general will benefit SMPC.

In general, this technique results in improvements not only to expressive cryptography, but to systems in general. For example, gateway-free Internet connectivity, provided by *TCPlp*, broadly benefits low-power wireless systems, beyond its applications to expressive cryptography. In applying this technique, the role of expressive cryptography is as a guide to find which resources are a significant bottleneck for expressive cryptography and which aspects of the underlying system tend to govern the performance of applications based on expressive cryptography. Once expressive cryptography draws one's focus to the appropriate aspects of the underlying system's design, the approach used to redesign the system is the same as in general systems research.

3.2 Designing Systems that Use Expressive Cryptography

The second class of techniques applies to designing applications, and systems that directly support applications, that are secured using expressive cryptography. Designing such applications and systems can be challenging because expressive cryptography can be very expensive. This section focuses on techniques that can make the overall application efficient, even it uses expressive cryptographic techniques that are inefficient. We present four such techniques below, focusing on how the system should use expressive cryptography. These techniques can be viewed as ways to design a system to make it "efficiently securable" via expressive cryptography.

3.2.1 Use Expressive Cryptography Rarely and Off of the Critical Path

The third system design technique is to use expressive cryptography *rarely* and *off of the critical path*.

Finding ways to use cryptography *rarely* is a widely-used technique. Hybrid-encryption-based techniques are an example of this. For example, TLS uses public-key cryptography during the TLS handshake, but uses exclusively symmetric-key encryption thereafter. Similarly, PGP uses public-key encryption to encrypt a symmetric key, and then uses the symmetric key to encrypt the actual message, which may be long. Such techniques naturally extend to the realm of expressive cryptography. For example, to encrypt a messages with key-policy attribute-based encryption (KP-ABE), one can first encrypt a symmetric key with KP-ABE, and then encrypt the actual message with the symmetric key. Clearly, finding ways to use expressive cryptography rarely can reduce costs for the application using it. It should be noted, however, that expressivity sometimes gets in the way. If the attributes change from one message to the next, then one cannot directly apply hybrid encryption with KP-ABE. Similarly, using hybrid encryption with homomorphic encryption, would preclude the ability to compute on encrypted data.

To understand the idea of using cryptography *off of the critical path*, we must consider the set of user-facing operations in a system. For example, a system for file storage and sharing may have user-facing operations like reading a file, writing a file, and sharing a file. In an IoT sensor application in which a sensor periodically collects sensor data (i.e., a "sense-and-send" application), collecting a physical reading from the transducers and sending over the network is a user-facing operation, since a user may be monitoring data as it is collected or may initiate collection of a particular sensor reading. Performing an expressive cryptographic operation as part of user-facing operations may degrade user-perceived performance because expressive cryptographic operations may have high *latency*. For example, a blockchain like Bitcoin requires *minutes* for a transaction to be accepted by the blockchain; as a result, a blockchain-based file system that requires a blockchain transaction to update a file would have a user-facing "write" operation that takes minutes to complete. Similarly, a single encryption with ABE might take *several seconds* or *minutes* to complete on a resource-constrained embedded device; encrypting each sensor reading in the aforementioned IoT application with ABE would, therefore, result in delays of seconds to minutes in obtaining a sensor reading.

Through careful system design, we can sometimes secure the entire system, including user-facing operations, with expressive cryptography, without using expressive cryptography in the critical path of user-facing operations. For example, Ghostor (Chapter 6), our data storage and sharing system, derives its integrity guarantees from a blockchain, yet ensures that all blockchain transactions are issued by the server in the background, separately from any user-facing operation. Similarly, JEDI (Chapter 7), our end-to-end encryption system for IoT, uses ABE with hybrid encryption, but is designed such that the encryption of the symmetric key can happen in the background, without being tied to any particular user-facing operation. This is because the attributes for a stream of data depend on the current time, which changes independently of sensor readings or user requests. Although these invocations of expressive cryptography can have a high latency, none of them directly affect the latency of any individual user operation.

Finding ways to use cryptography rarely can reduce the overall cost of expressive cryptography. Using if off of the critical path, however, cannot. For example, invoking ABE off of the critical path in JEDI can mask the long *latency* of invoking ABE, but it does not reduce CPU utilization or the problems associated with high CPU utilization (e.g., high power consumption, longer scheduling delays). The greatest benefits are derived from finding ways to use expressive cryptography *both* rarely and off of the critical path.

3.2.2 Make the Frequency of Expressive Cryptographic Operations Tunable

The fourth system design technique is to make the frequency of expressive cryptographic operations tunable. The idea is to find some aspect of the system's functionality or security that can be traded off, in return for being able to use expressive cryptography less frequently. This makes the system design inherently flexible, allowing one to tune the cryptographic overhead according to the needs of the application at hand.

We applied this technique in JEDI to control the frequency with which the protocol encrypts data with expressive cryptography. In JEDI, data producers (e.g., IoT devices) produce streams of data, and data consumers (e.g., users or applications) are granted access to streams of data. A user's access to data is usually granted with an *expiry time*, which is enforced cryptographically. In JEDI, the frequency with which a data producer must invoke expressive cryptography is tied to the granularity at which a data consumer's expiry time may be specified. For example, if the system is configured to allow a data consumer's expiry time to be specified as a timestamp with hour granularity, then a data producer must invoke expressive cryptography for a data stream once per hour; if the system is configured to allow a data consumer's expiry time to be specified as a timestamp with minute granularity, then a data producer must invoke expressive cryptography for a data stream once per minute. This adds a "knob" to the system configuration, allowing one to tune the overhead of expressive cryptography in JEDI according to the resource constraints of the application scenario by configuring the granularity at which expiry times may be specified. For the IoT sensing devices we considered in our study evaluating JEDI, we found that we can specify expiry times at one-hour granularity and achieve several years of battery life. To support IoT sensing devices that are even more energy-constrained, we could easily reconfigure the system (i.e., "turn the knob") to use coarser (e.g., six-hour granularity) expiry times. Conversely, if deploying JEDI on more powerful data producers without strict energy constraints, it may be feasible to support finer-grained (e.g., minute-granularity) expiry times.

We also applied this technique in Ghostor to control the frequency of blockchain transactions. Blockchain transactions are expensive not only because blockchains like Bitcoin have limited transaction throughput, but also because issuing a transaction on a blockchain has a monetary cost paid in cryptocurrency. Ghostor is a system for storing and sharing data that uses a blockchain to provide a *verifiable integrity* guarantee. Periodically, the server posts a small checkpoint to the blockchain. After interacting with a Ghostor server to access data, a user can wait until the next checkpoint and run a procedure to verify that the server correctly carried out operations before

the checkpoint. This means that if a Ghostor server posts checkpoints with a certain delay (e.g., once per three hours), then users will be able to detect any integrity violations on the server within that delay (e.g., within three hours). As a result, the frequency with which the server must issue blockchain transactions is tied to the delay within which users can detect integrity violations. This makes Ghostor, like JEDI, inherently flexible. For example, applications that can tolerate a longer delay to detect integrity violations can be supported with a lower-cost server configuration that posts checkpoints more rarely.

This technique can be understood as a generalization of using cryptography rarely. The efficiency gains come from being able to invoke expressive cryptography less frequently. The difference is that invoking expressive cryptography less frequently, with this technique, does not come for free—some aspect of the system's functionality or security (e.g., granularity of expiry times in JEDI or delay before integrity violations are detected in Ghostor) is traded off. Still, this technique can provide the needed flexibility to navigate the functionality-performance trade-off and tune the system's use of expressive cryptography according to the application scenario at hand. Such tuning can significantly impact efficiency; for example, changing the granularity of expiry times in JEDI from minute-level granularity to hour-level granularity reduces the cost of expressive cryptography by 60×, since there are 60 minutes in an hour. For application scenarios for which hour-level granularity is sufficient, these efficiency gains can be considered essentially free.

3.2.3 Identify and Use the Cheapest Cryptographic Primitive

The fifth system design technique is to identify and use the cheapest cryptographic primitive that provides the needed functionality and security. Due to the expressivity-efficiency trade-off (Section 2.2.2), this generally involves identifying the *least expressive* cryptographic tool that can support the desired application. Sometimes, the most widely known and studied cryptographic tools are more expressive than necessary for an application. Identifying a scheme that is less expressive yet expressive enough for the application can yield substantial performance gains.

For example, we applied this technique when choosing the expressive cryptographic scheme to use to encrypt data in JEDI. It is widely recognized that IoT applications stand to benefit from the complex access policies supported by Attribute-Based Encryption (ABE) schemes [355, 197]. Indeed, ABE supports the functionality that JEDI needs—it allows a data producer to encrypt a data stream according to a policy describing which data consumers should be able to receive it. In designing JEDI, we studied existing IoT systems, which do not encrypt data, to understand the kinds of policies that IoT systems must support. We found that ABE supports more general policies than existing IoT systems, unconstrained by cryptography, actually use. We identified that WKD-IBE [2], an encryption scheme that is less expressive than ABE, is sufficient to support the policies that IoT systems need. Yet WKD-IBE, being less expressive than ABE, is also more efficient than ABE—it is an order of magnitude faster to encrypt data and has significantly smaller ciphertexts. JEDI, by virtue of using WKD-IBE instead of ABE, inherits these performance benefits.

Note that, in general, no cryptographic scheme will exactly capture the needs of the application. Thus, it is often necessary to choose a scheme that is slightly more general. For example, even WKD-IBE, while closer to the needed functionality than ABE, still supports more general policies

than existing IoT systems unconstrained by cryptography need. As a result, it is not always clear that a cheaper, less expressive scheme that still provides the needed functionality and security even exists. In designing JEDI, we found that, while ABE supports more general policies than IoT systems need, prior expressive encryption schemes, like IBE and HIBE, are not expressive enough to support the needed policies. Furthermore, WKD-IBE is not an obvious choice, as it is less widely known than IBE, HIBE, or ABE. Identifying a cheaper encryption scheme, therefore, may be nontrivial, and it may even require custom design of a new cryptographic scheme specialized to the application at hand.

Ghostor's privacy design is not an application of this technique, but it is instructive to analyze it from the standpoint of this technique. Ghostor protects user privacy by delinking user identities from data accesses—that is, preventing the storage server from learning which users access a given data object. Some existing systems attain this property by achieving *obliviousness*—hiding from the storage server the data object to which each access corresponds. Unfortunately, this requires a linear scan over the data or multi-client variations of ORAM [31, 327], which are expensive. Ghostor takes a different approach—it hides the *user identity* corresponding to each access instead of which data object is accessed. This can be viewed as identifying the most efficient cryptographic primitive to achieve the desired security goal (i.e., identifying that ORAM is not necessary to delink user identities from accessed data); in this sense, Ghostor's privacy design is in the spirit of this technique. That said, we do not consider Ghostor's privacy design to be a direct application of this technique because it affects the security guarantees of the overall system—as we explain in Section 6.1.1, anonymity and obliviousness are different security guarantees that are largely orthogonal to one another. In contrast, using WKD-IBE instead of ABE in JEDI requires carefully choosing the system's *functionality* but does not affect security.

3.2.4 Develop Application-Specific Cryptographic Interfaces

The sixth system design technique is to develop application-specific cryptographic interfaces. Cryptographic schemes are often written with generic interfaces that allow them to be used in flexible ways. However, the way in which a *particular* system or application uses the scheme may allow an application-specific interface that admits additional algorithmic optimizations.

We applied this technique to JEDI to optimize the way in which it encrypts data. The WKD-IBE Encrypt function gives an algorithm to produce a ciphertext given public parameters, a list of attributes, and a message. We can represent this interface as $\textbf{Encrypt}(\mathsf{params}, \mathsf{attrs}, \mathsf{msg}) \to \mathsf{ct}$. In our JEDI protocol, a device repeatedly encrypts data with *similar sets of attributes*—adjacent encryptions usually only differ in one attribute. This allows us to develop an optimized encryption algorithm that uses an intermediate result from each encryption operation to accelerate the next encryption. Our new interface can be represented as $\textbf{Encrypt}(\mathsf{params}, Q, \mathsf{attrs}, \mathsf{msg}) \to \mathsf{ct}, Q'$, where Q is the intermediate result from the previous encryption and Q' is the intermediate result to use in the next encryption.[1] Computing the ciphertext from the intermediate result is 2–3×

[1]When we explain this technique in greater depth in Section 7.3.6.2, we split up the new **Encrypt** interface into two routines: **EncryptPrepared**, which outputs ct, and **AdjustPrecomputed**, which outputs Q'.

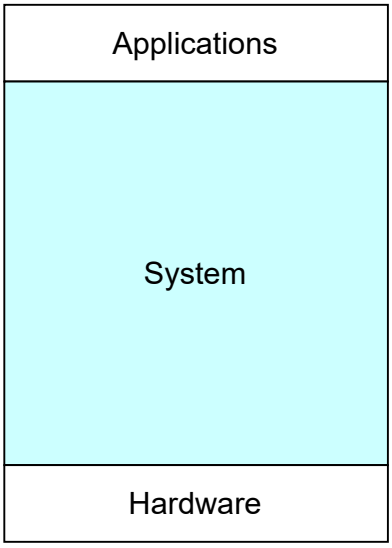


Figure 3.1: Classical system design: the system provides applications with access to the hardware.

faster than computing the ciphertext from scratch. At a high level, this particular optimization has similarities to caching—we are, in effect, caching the intermediate result Q and reusing it to accelerate the subsequent encryption.

Note that we are *not* proposing to develop new cryptographic schemes. While effective, developing a new cryptographic scheme would constitute a *cryptographic* technique, not a *system design* technique. Instead, we are proposing to develop new application-specific interfaces that provide faster algorithms to compute the same cryptographic objects (e.g., ciphertexts). In JEDI, for example, the ciphertexts resulting from our optimized Encrypt interface have exactly the same form as ciphertexts produced through the ordinary WKD-IBE Encrypt interface. The optimization merely provides a faster encryption *algorithm* for the existing WKD-IBE scheme, enabled by a new, JEDI-specific encryption interface to WKD-IBE.

3.3 When to Use Each Class of Techniques

The classical view of systems is as follows. Applications need to make use of computer hardware (i.e., computing resources) in order to function. The term *system* usually refers to the layer of software that manages computer hardware for users and their applications, providing applications with access to hardware and mediating their access as necessary to provide properties like protection and isolation. As shown in Figure 3.1, this can be visualized as a software stack with applications on top, hardware on the bottom, and the system in the middle.

On first glance, however, it is unclear exactly where expressive cryptography fits in this picture. On the one hand, expressive cryptography may be considered as part of the application; expressive cryptography consumes resources, and the techniques in Section 3.1 apply to systems that

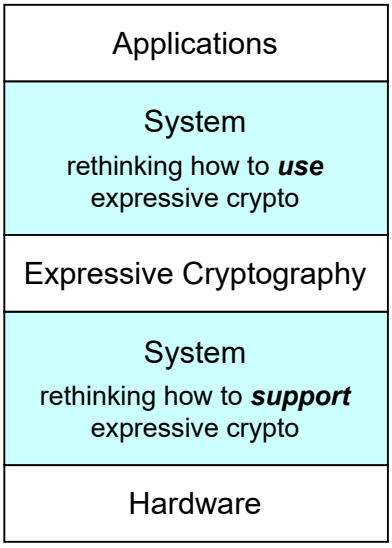


Figure 3.2: With expressive cryptography, there are two system layers: one providing applications with access to expressive cryptography, and another providing expressive cryptography with access to the hardware.

manage how expressive cryptography makes use of the underlying resources. On the other hand, expressive cryptography may itself be considered a resource; although expressive cryptography is not hardware, the techniques in Section 3.2 show how systems manage how applications make use of expressive cryptography. Clearly, expressive cryptography and system design are closely intertwined. We capture this with the software stack shown in Figure 3.2. Note that the classical software stack in Figure 3.1 remains relevant, as applications still need to access hardware for reasons other than using expressive cryptography. Thus, the two software stacks in Figure 3.1 and Figure 3.2 should be considered parallel stacks that co-exist side by side for applications that use expressive cryptography.

With the software stack in Figure 3.2, it is clear how each class of system design techniques is applicable. The first class of techniques (Section 3.1) applies to the underlying systems that expressive cryptography depends on—the lower system layer in Figure 3.2. The second class of techniques (Section 3.2) applies to systems that make use of expressive cryptography to support an application—the upper system layer in Figure 3.2. That said, some systems span *both* layers, and therefore benefit from *both* classes of techniques. As an example, consider systems that enable secure collaborative data analytics based on SMPC, such as SMCQL [38] and Conclave [462]. To minimize the overhead of SMPC, these systems should do two things. First, they should execute SMPC as efficiently as possible. The first class of techniques addresses this. Systems like MAGE can allow them to do so, by reducing the memory-related performance overheads of SMPC. Second, they should use SMPC in as efficient a way as possible. The second class of techniques addresses this. For example, both SMCQL and Conclave have cryptographic planners

(Section 8.1.2) that aim to produce executions plans that use SMPC in a minimal way, relying on non-cryptographic computation wherever possible. An individual research project may focus on one of these layers, and therefore draw primarily from only one class of techniques—for example, the SMCQL and Conclave publications focus primarily on the respective systems' cryptographic planners. But the *overall system* has both a cryptographic planner and execution engine, and therefore benefits from *both* classes of techniques. Research projects that take a holistic view of the system may also draw from both classes of techniques. For example, JEDI not only presents techniques for using an expressive encryption scheme in an efficient way (second class of techniques), but also includes hand-written assembly for executing the expressive encryption scheme (first class of techniques). Both aspects of JEDI's design contributed to performance improvements compared to a naïve baseline design.

The next four chapters demonstrate the validity of these techniques through the design, implementation, and evaluation of systems designed using them. The chapters are organized roughly according to the order of the techniques above. Chapter 4 describes MAGE, which can be understood as an application of the first technique (Section 3.1.1). Chapter 5 describes *TCPlp*, which is an application of the second technique (Section 3.1.2). Chapter 6 describes Ghostor, which is based on the third and fourth techniques (Section 3.2.1 and Section 3.2.2). Chapter 7 describes JEDI, which applies all of the techniques except the first, in particular including the fifth and sixth techniques (Section 3.2.3 and Section 3.2.4).

Chapter 4

Supporting Secure Computation with Nearly Zero-Cost Virtual Memory

This is the first of two chapters exploring the techniques in Section 3.1. We focus in this chapter on tools for computing on encrypted data, like Secure Multi-Party Computation (Section 2.1.3.1) and Homomorphic Encryption (Section 2.1.3.2). As described in Sections 2.1.3 and 2.2, these tools enable compelling applications like secure collaborative data analytics, but a significant obstacle is the memory overhead of the underlying cryptography.

This chapter presents the design, implementation, and evaluation of MAGE, an execution engine for computing on encrypted data. By applying the technique from Section 3.1.1, MAGE addresses the high memory overhead of these tools, efficiently running encrypted computations that do not fit in memory. MAGE significantly outperforms the OS virtual memory system, and in many cases, runs encrypted computations that do not fit in memory at nearly the same speed as if the underlying machines had *unbounded physical memory* to fit the entire computation.

4.1 Introduction

As described in Section 2.1.3, a variety of expressive cryptographic tools exist for directly protecting computation. We use the term *Secure Computation (SC)* to refer to the subset of those tools that enable computation on encrypted data, like Secure Multi-Party Computation (SMPC) and Homomorphic Encryption (HE). Recently, the use of SC in industry has burgeoned. Companies offer services based on SC [241, 147, 122, 98, 372, 270] (from secure collaborative learning to decentralized authentication and custody), large financial enterprises have added SC-based products [373], and cryptocurrencies secure billions of dollars with SC [506].

As described in Section 2.2, SC not only has high CPU overhead, but also requires high memory usage and, in the case of SMPC, high network usage. For example, a 64-bit integer, which requires only 8 B of memory when computing in plaintext, takes up 1 KiB of memory when using garbled circuits, a type of SMPC. Efficiently running SC requires careful attention to not only CPU efficiency, but also memory and network demands.

CPU overheads can be reduced using hardware accelerators (e.g., GPUs, FPGAs) or specialized hardware (e.g., AES-NI). Network bandwidth continues to grow exponentially according to Nielsen's Law [357], and recent cryptographic improvements have relaxed network bandwidth demands for some SC protocols [85, 109]. But memory management remains problematic. Many recent cryptographic systems based on SC report that SC systems quickly run out of memory [462, 378, 515, 516]. Once they do, the computation becomes prohibitively slow because the OS inefficiently swaps the large memory footprint to secondary storage. For example, the authors of Conclave [462] report that Obliv-C, an SMPC framework, can run a join on only 30,000 records before running out of memory, and state that SMPC "in practice only scales to a few thousand input records." Similarly, Senate [378], a secure collaborative analytics engine based on SMPC, can run a 16-party private set intersection on only 10,000 integers per party.

In this context, we address the research question: **Can SC execute efficiently when it does not fit in memory?** We answer this in the affirmative with our system MAGE, an execution engine for SC. MAGE stands for **M**emory-**A**ware **G**arbling **E**ngine, but it is not limited to garbled circuits.

A natural starting point for MAGE is to specialize the OS page replacement policy to SC workloads. Unsurprisingly, this design suffers from some of the same pitfalls as classic virtual memory systems. Pages may not be fetched until a page fault occurs, requiring multiprogramming to keep the CPU busy [142]. Furthermore, classic page replacement algorithms perform poorly on some workloads [41], and a policy specialized to SC would likely be no different.

To mitigate these issues, we observe that SC is inherently *oblivious*. In particular, many SC protocols have *no data-dependent memory accesses*. This is because an SC protocol must not leak any information about the data contents via its memory access pattern. Our key insight in MAGE is that SC's inherent obliviousness allows us to calculate the access pattern for any computation *in advance* and use it to manage memory in a fundamentally more efficient way than classic OS paging. This is an application of the technique from Section 3.1.1 to manage memory according to the structure of the computation (i.e., its obliviousness). Unlike paging, which typically responds to page faults *reactively*, MAGE can *proactively* produce a memory management plan based on the program's memory access pattern. To highlight this distinction, we call our approach *memory programming* and the resulting plan a *memory program*. MAGE preplans the exact sequence of memory-storage transfers to issue at runtime, given a target memory consumption. Enabled by memory programming and the compute-to-memory ratio of SC workloads, MAGE executes certain SC programs that do not fit in memory at nearly in-memory speeds, as if memory were unbounded—in effect, virtual memory at nearly zero cost.

To understand the power of MAGE's preplanning based on SC's obliviousness, consider Belady's theoretically optimal paging algorithm (MIN) [41]. MIN, being a clairvoyant algorithm, is not realizable in the classic formulation of paging; it is typically used as a point of comparison to other realizable heuristics. But in the context of memory programming, MAGE can use MIN directly, because it knows the access pattern in advance. Memory programming allows MAGE to use an algorithm that is well-grounded in theory, instead of a heuristic (e.g., LRU or LFU) that sometimes performs poorly.

Yet memory programming also raises the bar for possible memory management strategies. For example, although MIN is an optimal paging algorithm, it unfortunately does not produce an

optimal memory program. The reason is that MIN, like other demand paging algorithms, brings a page into memory only at the moment it is needed, thereby causing the program to stall while transferring the page. We can overcome this by leveraging SC's obliviousness once again, to prefetch according to the access pattern (i.e., with no false positives or false negatives) so that the program never stalls.

At its core, our approach to memory management is quite simple: MAGE optimizes storage bandwidth by evicting pages using MIN, and optimizes latency via prefetching and asynchronous eviction. Whereas classic paging algorithms typically rely on heuristics and empirical observations of what works well in practice [80], our memory programming approach is simple, well-grounded, robust, and performant.

While conceptually simple, the above strategy is challenging to instantiate efficiently. The reason is that MIN requires the entire memory access pattern to be materialized at once; it cannot be applied in a streaming fashion. Using Intel Pin [325], we found that an SC workload that runs in under an hour can issue *trillions* of memory accesses. Thus, materializing the access trace could require *terabytes* of space.

To address this, we leverage the strong precedent for using DSLs to specify SC programs [214, 459]. MAGE's planner represents the program as a bytecode recording higher-level operations specified in the DSL program. This is more succinct than recording individual memory accesses. For example, consider a program that adds two integers using garbled circuits, an SMPC protocol. Garbled circuits support only AND and XOR operations on encrypted *bits*, so the integer addition is ultimately decomposed into encrypted AND and XOR operations, each of which comprises many memory accesses. Yet, MAGE records the entire addition operation as a *single* entry in the bytecode. This works well because most of the addition operation's memory accesses are "uninteresting"—they are accesses to temporary variables (e.g., on the stack) that fit easily in memory, or to SC protocol state that should remain in memory for the entire program. The only consequential accesses for memory management—reading the two input integers and writing the output integer—are captured in the single entry MAGE records.

Once MAGE allows SC to efficiently expand beyond the physical memory limit, another limited resource (e.g., storage/network bandwidth or CPUs) of a single machine could become the bottleneck. Thus, we design MAGE to support *parallel* SC execution across multiple network flows, CPU cores, or machines. To do so, we observe that a distributed memory programming model allows SC to be parallelized in this way, without requiring MAGE's planner to reason about threads executing concurrently in the same address space.

Finally, we aim to support a variety of applications and protocols, including new ones that may emerge in the coming years. The challenge is that different SC protocols may be very different cryptographically and may support different operations efficiently. Fortunately, our memory programming approach allows us to build MAGE entirely in userspace on a Linux system, helping to make MAGE *extensible* to new applications and protocols. We carefully design a layered architecture for MAGE so that the DSL, bytecode, and interpreter can be extended for new SC protocols.

We implemented MAGE in C++ and apply it to two SC protocols: (1) garbled circuits, a type of SMPC, and (2) CKKS, a type of HE. We evaluated MAGE using 10 workloads, sized such

that they do not fit in memory. MAGE outperforms the operating system's virtual memory for all 10 workloads, and outperforms it by 4–12× for 7 of them. Additionally, MAGE executes all 10 workloads at within 60% of in-memory speeds, and runs 7 of them at within 15% of in-memory speeds.

Even with our techniques, SC remains orders of magnitude slower than plaintext computation due to CPU and network overheads. That said, various applications like federated data analytics [38, 378], coopetitive machine learning [515], and privacy-preserving recommendation [358] *require* SC. Due to privacy constraints, running these applications in plaintext is not an option. By bringing memory management overhead for SC to nearly zero, MAGE helps make SC more practical and potentially enables more SC-based applications.

4.2 Secure Computation Background

In this section, we augment the background provided in Section 2.1.3 with details that will help the reader understand our system MAGE.

4.2.1 Circuit Representation

As explained earlier, SC is inherently oblivious, meaning that any function f computed using SC cannot have data-dependent memory accesses. Thus, it is natural to describe the function f as a circuit C [332, 230, 99, 134]. C is a combinational circuit that accepts the arguments to f as inputs and produces the result of f applied to those arguments as its output. We write $C = (W, G)$, where W is a set of wires and G is a set of gates. Each *wire* represents a datum whose type is the unit of computation in the SC scheme (in most cases, it is the information stored in a single ciphertext). We denote the subset of W storing C's input as I, and the subset of W storing C's output as O. Each *gate* represents a computation supported by the SC scheme. We will typically assume that each gate has exactly one output wire, and that each $w \notin O$ is the input wire of at least one gate. Thus, $|W| = |G| + |I|$.

The particular data types represented in the wires and the types of supported gates depend on the particular SC scheme of interest. For the CKKS homomorphic encryption scheme [115], each wire represents a *vector of real numbers* and each gate represents an element-wise *addition or multiplication* of those vectors. For garbled circuits [496], each wire represents a *single bit* and each gate represents a binary *AND operation or XOR operation* on those bits. Other SC schemes can be similarly formulated this way. Below, we explain CKKS and garbled circuits in greater depth.

4.2.2 CKKS Homomorphic Encryption

In the CKKS scheme [115], each ciphertext encodes a vector of real or complex numbers (stored with limited precision). Given ciphertexts $c_1 = \mathsf{Enc}(\vec{v_1})$ and $c_2 = \mathsf{Enc}(\vec{v_2})$, one can compute $\mathsf{Enc}(\vec{v_1} + \vec{v_2})$ and $\mathsf{Enc}(\vec{v_1} \circ \vec{v_2})$ (where $\circ$ is element-wise multiplication). The dimension of each

vector depends on parameters chosen during key generation. Each ciphertext is assigned a level, which is a nonnegative integer. When performing the element-wise multiplication operation, both input ciphertexts must have the same level; the level of the output ciphertext is one less than the level of the inputs. Performing element-wise addition does not reduce the ciphertext level the way element-wise multiplication does. A ciphertext at level 0 cannot be used for element-wise multiplication. The maximum level of a ciphertext depends on the parameters chosen during key generation. While one can run a bootstrapping procedure to increase the level of a ciphertext, it is very expensive, and therefore not implemented by all libraries.

4.2.3 Garbled Circuits

Yao's garbled circuit protocol [496] (referred to simply as *garbled circuits*) allows two parties, called the *garbler* and the *evaluator*, to jointly compute a function f over their private inputs x_1 and x_2. The protocol requires f to be represented as a *boolean circuit* C. Unlike CKKS, there are no restrictions on C's depth. However, *both* parties have to execute the circuit.

First, the two parties run a protocol called *oblivious transfer* to obtain the (encrypted) wire values for their inputs without revealing their inputs. Then the garbler encrypts C in a special way called *garbling* to obtain $\widetilde{C}$, called a *garbled circuit*. The process of garbling is analogous to executing the circuit; a gate cannot be garbled until the (encrypted) values of both input wires are obtained, and garbling a gate produces, as a side effect, the (encrypted) value of the output wire. Then, the garbler sends $\widetilde{C}$ to the evaluator. The evaluator executes the circuit, executing each gate using the gate's garbled information in $\widetilde{C}$. Finally, the two parties communicate to decipher the plaintext values of the output wires.

If the parties would like to repeat the computation again with different inputs, they must re-garble C. It is insecure to reuse the same garbled circuit $\widetilde{C}$ with different sets of inputs.

More comprehensive explanations of garbled circuits, their underlying cryptography, and their state-of-the-art optimizations are available in other resources [398, 53, 494].

4.2.4 Efficiently Executing Circuits

In this section, we give background on existing techniques for efficiently executing cryptographic circuits. Although many of these techniques were developed for garbled circuits, they mostly apply to homomorphic encryption as well.

4.2.4.1 Naïve Baseline

Early garbled circuit systems, like Fairplay [332], JKS [255], and PSPW [376], allocate memory for all wires and store the entire garbled circuit in memory. The memory overhead is $\mathcal{O}(|W|+|G|)$. Because, for a well-formed circuit, $|G|+|I|=|W|$, this is equivalent to $\mathcal{O}(|W|)$.

4.2.4.2 Pipelining Garbling and Evaluation

After the garbler garbles a gate to include in $\widetilde{C}$, the garbler does not use that gate's garbled data. Similarly, once the evaluator evaluates a gate, it never again uses that garbled gate. Based on this observation, the HEKM system [230] operates without keeping the entire garbled circuit in memory, as follows. The garbler and the evaluator first agree on an order in which to execute the gates in C. Then, the garbler garbles each gate and streams the garbled gates to the evaluator, who evaluates the gates in the same order. In this way, all gates are garbled and evaluated, without materializing the full set of garbled gates at any one time. Because space is allocated for all wires in the circuit, the memory overhead is still $\mathcal{O}(|W|)$.

4.2.4.3 Reclaiming Wire Memory

When executing a circuit, one can discard the memory for a wire once all gates it feeds into have been executed. Only wires whose values have been computed and will be used in the future—the *live* wires—must be kept in memory. The KSS system [288] takes advantage of this by dynamically attaching a reference count to each wire; PCF [287] statically calculates when to reuse wire memory. Using interpretation techniques developed in PCF [287] and refined in Frigate [348], not even the plaintext circuit is materialized in memory. TinyGarble [436], EMP-toolkit [473] (for semi-honest SMPC), and EVA [134] also use variants of this technique. With this optimization, the memory demand is $\mathcal{O}(w)$, where w is the size of the largest set of live wires when executing the circuit. MAGE builds on this line of work by exploring how to efficiently swap to storage when w wires do not fit in memory.

4.3 Memory Overhead of Secure Computation

First, we discuss the memory overhead of SC. Then, we discuss the memory overhead for collaborative applications.

4.3.1 Analysis of the Memory Demand

The size of the circuit, for a computation, is proportional to the size of the computation. But in many cases, the memory demand is substantially smaller than the circuit size; only w wires need to be stored, where w is the size of the largest set of live wires when executing the circuit (Section 4.2.4.3).

In practice, circuits are often described in a programming language [214, 459] and gates are executed in the same order as the program is interpreted. In this execution order, live wires correspond to in-scope variables in the program. Thus, the memory usage of running an SC program has the same order of growth as running the same algorithm in plaintext.

The memory cost of SC lies in the constant factors. When executing a secure computation protocol, the wire values are encrypted. Thus, a key parameter is the expansion factor of the encryption. In garbled circuits using a 128-bit block cipher, including state-of-the-art optimizations

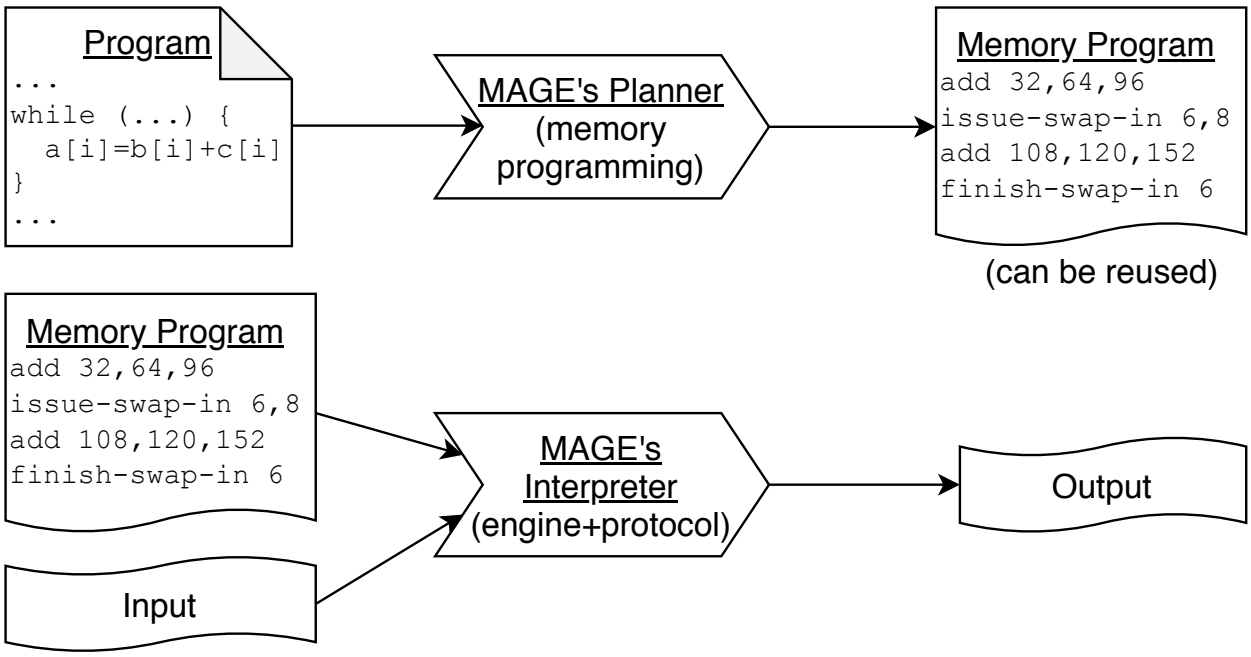


Figure 4.1: Overview of MAGE. It consists of two phases: planning (top) and execution (bottom).

(Point-and-Permute [40], Free XOR [284], Half Gates [504], and Fixed-Key Block Cipher [45, 206]), each wire value is 16 bytes. Each wire represents only 1 bit of plaintext, so this is a 128× expansion factor. For CKKS, ciphertexts at higher levels are larger than ciphertexts at lower levels. For the parameters we used in our evaluation, each ciphertext is hundreds of kilobytes and encodes a vector of dimension up to 4,096.

4.3.2 Scaling Collaborative Applications

SMPC supports *collaborative applications* over secret data, such as federated data analytics [38] and cooperative machine learning [346]. A common technique to reduce SMPC's overhead is to use SMPC in a *minimal way*. For example, some approaches aim to use SMPC for only a small part of the overall computation [38, 322, 260, 462, 515]. Others carefully choose algorithms that can be executed efficiently in SMPC or use approximations that incur less overhead [346, 397, 345]. But even with these approaches, the SMPC computation often has high memory demands [378]. Thus, it remains important to efficiently execute SMPC computations that do not fit in memory.

4.4 System Overview

SC workloads are oblivious by nature. Thus, MAGE can work out the program's memory access pattern in advance, and use this information to produce a memory management plan, called a

memory program, tailored to the particular access pattern. Importantly, obliviousness is not merely an artifact of certain existing SC schemes; it is inherent to SC. Otherwise, an adversary could potentially infer information about secret data based on the memory access pattern.

To support this paradigm, MAGE's workflow has two phases, as shown in Figure 4.1. An SC application is written in a DSL internal to C++. MAGE's planner unrolls the DSL code to produce a bytecode, and then performs transformations on the bytecode to produce a memory program. In MAGE, the memory program is a bytecode that includes *swap directives* describing when to transfer data between storage and memory. Finally, the memory program is given to MAGE's interpreter, which executes it using the SC protocol.

For multi-party protocols, the parties run separate instances of MAGE's interpreter. In the case of garbled circuits, garbled gates are streamed from the garbler to the evaluator, as described in Section 4.2.4.2. Both the garbler and evaluator use MAGE to follow a memory program and run with constrained memory.

Our approach of including swap directives in the memory program relies on the planner knowing how much memory will be available at runtime. An alternative approach is for memory programs to be agnostic to the amount of available memory. This would add runtime overhead, as MAGE's interpreter would need to decide which pages to evict. In contrast, our approach moves this overhead to the planning phase, keeping the execution phase as lightweight as possible.

4.4.1 Address Translation

The application programmer should not have to manage paging, so it is natural to write DSL programs in a virtual address space that is, in effect, infinitely large. Central to designing MAGE is deciding at which point in Figure 4.1 to translate this address space into a physical address space that fits in RAM.

One possibility (which MAGE does not use) is to perform address translation at runtime, using standard operating system mechanisms for prefetching and address translation. At runtime, swap directives in the memory program would ask the operating system to page parts of the virtual address space out to storage or in to RAM. Unfortunately, the existing way for a Linux process to do this—the `madvise` system call—is too limited. As of Linux 5.10, pages brought into RAM using the `MADV_WILLNEED` hint are not mapped in the page table, so a minor page fault is incurred on the first subsequent access. Similarly, the `MADV_PAGEOUT` hint merely marks pages as inactive; it does not swap out pages immediately.

In contrast, MAGE does not rely on OS address translation for demand paging. MAGE's engine moves data between memory and storage via explicit I/O operations, so that its resident set size never exceeds the available RAM. At the surface, this is similar to buffer management in a DBMS. But unlike a DBMS, MAGE's planner can be viewed as solving an address translation problem in advance. The DSL variables declared by the programmer exist in a *MAGE-virtual address space*, and the final memory program output by the planner references data (i.e., wire values) in a *MAGE-physical address space* that fits within RAM. MAGE's planner creates these address spaces and performs their translation in software during the planning phase. It includes swap directives in the memory program so that the interpreter does not run out of RAM.

To avoid confusion, we will refer to the addresses created by the OS and sent over the memory bus as *OS-virtual addresses* and *OS-physical addresses*. At runtime, MAGE's interpreter stores the program's memory in an array, and each MAGE-physical address in the memory program is treated as an index into this array. Thus, MAGE-physical addresses roughly correspond to the OS-virtual addresses of MAGE's interpreter.

MAGE's approach to address translation has several advantages. First, unlike an `madvise`-based approach, MAGE's planner has nearly complete control over when pages are brought into memory and evicted to storage. Second, by translating addresses in the planner, MAGE avoids address-translation-related overheads at runtime. In contrast, relying on OS address translation would mean minor page faults, page table updates, and TLB invalidations at runtime.

MAGE's approach also has a few drawbacks, however. First, the planning phase takes longer because MAGE's planner must translate all addresses in software. Second, memory programs are considerably larger because they must contain not only swap directives, but also a copy of the program translated to operate on MAGE-physical addresses. In particular, the memory program's length is proportional to the program's execution time because a variable local to a function or loop could be assigned different physical addresses each time the function is called or on each iteration of the loop.

Overall, we felt that the advantages of this design outweighed its drawbacks. Longer planning times seemed reasonable because planning can happen offline and the resulting memory program can be used repeatedly. The larger memory program size was an acceptable trade-off because MAGE's planner materializes an unrolled form of the program anyway to run Belady's algorithm. Meanwhile, MAGE's planner is afforded nearly full control of page eviction and replacement and MAGE's runtime overheads remain relatively small.

4.4.2 Bytecode Representation

Recall that MAGE's planner expresses the program as an unrolled (branch-free) bytecode, and performs transformations on it to compute the memory program bytecode. What operations should the bytecode instructions support?

One possibility would be for the bytecode to describe low-level operations similar to those supported by a CPU, excluding control flow instructions. Unfortunately, such a bytecode includes the raw memory trace of the program, which, as discussed in Section 4.1, can be impractically large.

One alternative, used by PCF [287] and Frigate [348][1] (but not MAGE), is to have each instruction correspond to a gate in the circuit C being executed. This approach would require a *protocol driver* in MAGE's interpreter that executes each gate using the SC protocol. To understand why this is inefficient, consider garbled circuits, for which gates are binary and wires represent bits. The programmer specifies the circuit in terms of operations on high-level types such as integers, which are then compiled into bit-level operations. Thus, each time the program performs a high-level

[1] Unlike MAGE, these systems also include control flow operations.

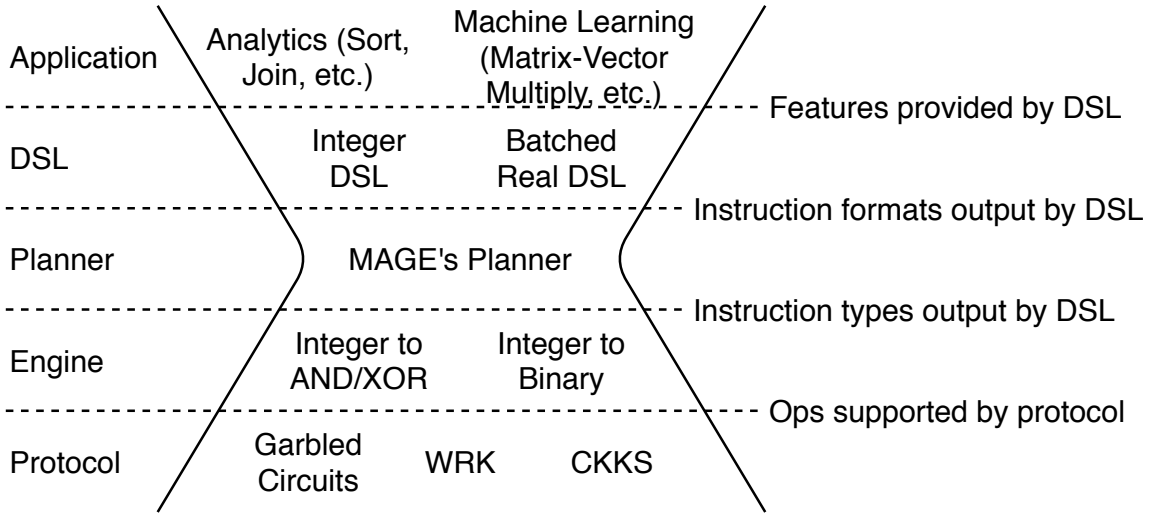


Figure 4.2: MAGE's envisioned ecosystem, with planning as the narrow waist.

operation (e.g., adding two integers), the same subcircuit (e.g., describing integer addition in terms of binary gates) is repeated in the bytecode.

To eliminate this repetition, MAGE has each instruction describe a high-level operation directly. This requires not only a *protocol driver*, but also an *engine* in MAGE's interpreter that expands each instruction into the relevant subcircuit at runtime. MAGE's planner does not need to materialize the subcircuits because wires internal to the subcircuits are very short-lived and therefore can be ignored.

4.4.3 Ecosystem and Extensibility

An important consideration in MAGE's design is to be applicable to a range of SC protocols. For example, garbled circuits and homomorphic encryption (CKKS) have quite different computation models, yet we show how MAGE captures both. MAGE's envisioned ecosystem can be understood as a set of layers with a narrow waist, as shown in Figure 4.2. The narrow waist is MAGE's planner; MAGE's core planning algorithms can be used with a variety of applications and interpreters.

MAGE's interpreter has two layers. The upper layer, called the *engine*, decomposes each instruction into a subcircuit of gates supported by the target SC protocol (Section 4.4.2). The lower layer, called the *protocol driver*, evaluates gates with the SC protocol. For example, when using a protocol that supports only binary AND and XOR operations (e.g., garbled circuits), one must use an engine that decomposes each instruction into a circuit of AND and XOR gates. In contrast, when using a protocol that supports all types of binary gates (e.g., TFHE [116]), one can use an engine that uses all types of gates.

One must choose compatible implementations at each layer. For example, once one has selected an SC protocol, one should choose an engine that executes each instruction using operations

supported by that protocol. Then, one should select a DSL that outputs instructions that the chosen engine understands. Finally, one must write the application in that DSL.

MAGE's planner, however, is universally compatible, allowing it to be the "narrow waist" of the ecosystem. The first reason is that MAGE's planner does not have to understand what each instruction *does*, only what memory it accesses. Thus, even if a new instruction is introduced into a DSL, extending a header file to specify its format (which includes which fields are memory addresses) is enough for the planner to understand that instruction. The second reason is that MAGE's planner does not introduce any new instructions except for swap directives, which all engines understand. Thus, if an engine understands the instruction types output by MAGE's DSL, then the engine will also be able to interpret the planner's output (i.e., the memory program).

A number of frameworks and DSLs for SC [214, 459] aim to make it easier for non-SC-experts to use SC. In contrast, MAGE is an efficient SC execution engine; its DSLs are not necessarily geared toward non-experts, do not optimize the resulting circuit, and might expose low-level SC operations. We discuss how these frameworks fit into Figure 4.2 in Section 4.9.

4.5 Engine

MAGE's execution engine is an interpreter for the final memory program. First, it allocates an array to store the program's data. Each MAGE-physical address is an index into this array. To execute an instruction, MAGE reads the instruction's arguments from this array, makes calls to the protocol layer to compute the output, and writes the output back to the array. Each instruction in the memory program references its input and output data directly by MAGE-physical address; the engine sees no MAGE-virtual addresses. Some instructions, such as those requesting pages to be transferred between storage and memory, are handled directly by the engine, without calling the protocol. We call such instructions *directives*.

4.5.1 Parallel/Distributed Engine

SC is resource-intensive, so it is natural to scale SC by executing the protocol in a distributed fashion across *multiple CPU cores* or *multiple machines*. The multiple-machine case is useful to overcome resource constraints associated with a single machine such as limited CPU cores, limited storage I/O, or, in the case of SMPC, limited network bandwidth. This is different from having multiple parties in SMPC. Here, we are parallelizing a single trust domain—for example, a single logical party in SMPC may execute using multiple machines.

MAGE's engine supports distributed execution across multiple *workers*. Each worker is a thread of computation, running MAGE's engine, operating on its own memory region (a MAGE-physical address space). Workers differ from OS processes as follows: (1) each worker contains exactly one thread, (2) workers are not necessarily isolated by hardware such as an MMU—multiple workers in a MAGE computation could, in principle, run within the same process, and (3) memory is statically partitioned among the workers.

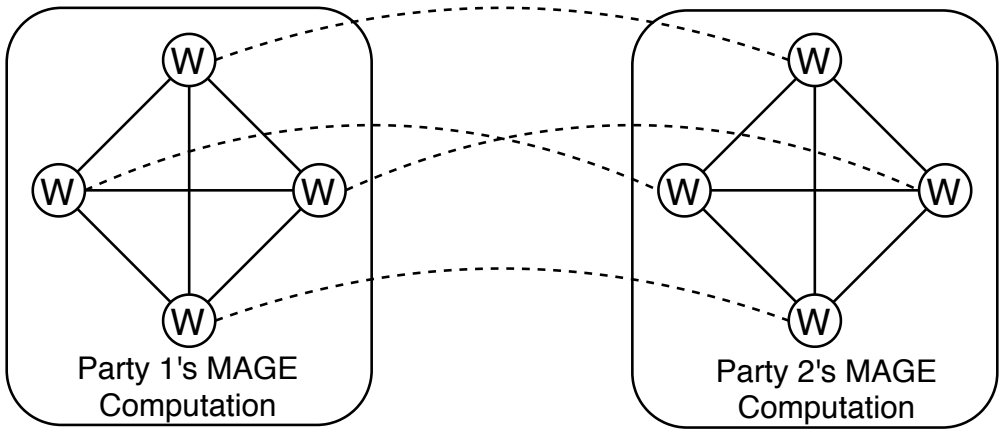


Figure 4.3: Example of distributed SMPC with MAGE. Workers are denoted as circles with W. Solid lines indicate connections managed by MAGE's engine; dashed lines indicate connections managed by the protocol driver.

MAGE's planner does not automatically infer how to parallelize the computation. Rather, the programmer writes DSL code in a distributed memory model, explicitly indicating asynchronous network operations to transfer data among the different workers. The resulting memory program bytecode contains *network directives* that the engine interprets. Similarly, the protocol driver must be written to function properly when the computation is distributed over multiple workers.

Programs for MAGE are parameterized by the Worker ID. MAGE's planner is run once *for each worker*. To generate the memory program for a worker, the planner processes only the accesses for that worker—it does not need to consider other workers' accesses, because each worker can only access its own memory region. Thus, the workers' memory programs can be generated independently and in parallel.

Using a distributed memory model provides two benefits. First, it allows MAGE to be agnostic to whether workers are placed on a single machine or across multiple machines. Second, it guarantees that the access pattern for each region of memory consists of a single well-defined sequence, simplifying planning. To ease the difficulty of explicitly specifying network transfers, one can build easier-to-use DSL libraries for common communication patterns (e.g., our implementation provides a `ShardedArray<T>` abstraction).

4.5.2 Distributed SMPC

Some SC protocols, like SMPC, require interaction over the network between mutually distrusting parties. For such protocols, each party runs a separate MAGE computation, with its own set of workers. Whereas the MAGE engine handles *intra-party communication* between workers in the same party, the protocol implementation handles *inter-party communication* among workers in different parties. The inter-party topology is up to the protocol driver; our protocol driver for garbled circuits uses a one-to-one inter-party topology (Figure 4.3).

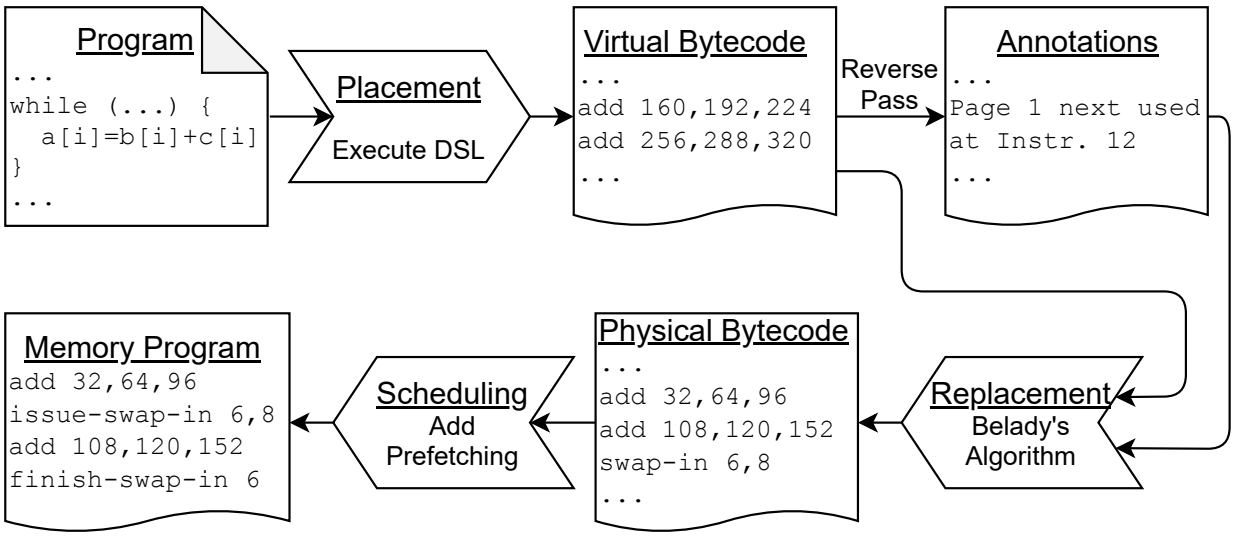


Figure 4.4: MAGE's planner's workflow, with its three stages.

4.6 Planner

Our memory programming approach is to calculate the memory access pattern in advance and use it to preplan memory management. One can potentially preplan the following:

- **Placement.** How should we divide up a circuit into pages?
- **Ordering.** In what order should we evaluate the gates in the SC circuit to result in the best memory behavior?
- **Scheduling.** When should pages that will be used in the future be swapped in from storage?
- **Replacement.** How should we choose pages to evict when making room for pages from storage?

MAGE produces an approximate solution, using a heuristic for placement and optimizing scheduling and replacement. Note that MAGE does not optimize ordering; it evaluates gates in the order implicit in the DSL program for the circuit.[2]

4.6.1 Organization of the Planner

We organize MAGE's planner into stages (Figure 4.4):

[2]Optimizing ordering may be NP-hard [301]. A system that does so would be very powerful—for example, it would automatically block a loop join or tile a matrix multiplication. It is beyond the scope of this work.

1. **Placement.** This stage accepts a DSL program and organizes wires into MAGE-virtual pages. It outputs instructions referencing wires by MAGE-virtual address.

2. **Replacement.** This stage adds instructions to swap pages to/from storage, deciding which pages to evict. It outputs instructions referencing wires by MAGE-physical address.

3. **Scheduling.** This stage moves swap instructions within the instruction stream and relocates wires to mask the latency of moving data between memory and storage.

For a parallel/distributed program, MAGE's planner is invoked separately for each worker, with separate MAGE-virtual and MAGE-physical address spaces. Network directives in the program transfer data among those address spaces.

MAGE's planner does not benefit from MAGE's memory programming techniques, so it is important that planning does not consume an unreasonable amount of memory. We keep the planner's memory usage lightweight by (1) writing/reading the intermediate bytecodes to/from files instead of keeping it all in memory, (2) designing the DSLs to be lightweight, and (3) keeping track of pages instead of individual bytes.

4.6.2 First Stage: Placement

MAGE's placement module is, in effect, a page-aware memory allocator for the DSL. It unrolls the DSL, allocating space for each variable and intermediate value in the MAGE-virtual address space. It outputs a bytecode for the program in which each variable is referenced by its MAGE-virtual address.

4.6.2.1 Unrolling the DSL Code

MAGE's DSLs are internal to C++. This means that the DSL is a set of convenient C++ APIs to specify the program's behavior, often involving operator overloading. The program is specified as a C++ function that uses these APIs.

Figure 4.5 shows a program that solves Yao's Millionaire's problem [495]. `Integer<width>` describes an Integer datum with the specified width in bits. `Bit` is an alias for `Integer<1>`.

MAGE's planner does not parse the DSL program's source code or manipulate its AST. Instead, it simply calls the C++ function containing the DSL program. As the DSL code executes, it produces a bytecode describing the computation. For example, the overloaded + operator for `Integer` emits an `Add` instruction in the output bytecode; it does not actually add integers using secure computation. Each output instruction references its operands by MAGE-virtual address. Thus, the DSL (e.g., the `Integer` class) calls MAGE's placement module to allocate memory in the MAGE-virtual address space for intermediate results, including those stored in variables.

For example, see Figure 4.5. On the `mark_input` and `>=` operations, an allocation request is made to MAGE's placement module to obtain a MAGE-virtual address, and an instruction is emitted to perform that operation (obtain input or integer comparison) and store the result at that MAGE-virtual address. Once an `Integer`'s destructor is called, or if an `Integer` is reassigned to

```
void millionaire(const ProgramOptions& args) {
    Integer<32> alice_wealth, bob_wealth;
    alice_wealth.mark_input(Party::Garbler);
    bob_wealth.mark_input(Party::Evaluator);
    Bit result = alice_wealth >= bob_wealth;
    result.mark_output();
}
```

Figure 4.5: Example code in an Integer-based DSL internal to C++ to solve Yao's Millionaire's problem.

a new MAGE-virtual address, a deallocation request is made to MAGE's placement module for the MAGE-virtual address previously held by that `Integer`.

For a parallel/distributed program, the worker ID and total number of workers are provided via the `ProgramOptions` structure. The C++ code can branch on these variables, to have each worker operate differently and exchange data appropriately to perform the parallel/distributed computation.

Each `Integer` object contains only the MAGE-virtual address of its contents; other attributes, such as width, are template arguments and do not consume memory. Thus, `Integers` and other DSL-provided data types are typically smaller than the encrypted data items they represent. For example, a 32-bit integer encrypted for the garbled circuit protocol is 1 KiB in size, whereas an `Integer<32>` object used during planning is just 8 B (a single MAGE-virtual pointer). This helps keep the memory cost of the planning phase small.

4.6.2.2 Memory Allocation Strategy

When MAGE's placement module allocates memory for a variable, it ensures that the variable is contained in a single MAGE-virtual page; a variable must never straddle two pages. The reason is that two adjacent MAGE-virtual pages may not be adjacent in the OS-virtual address space at runtime.

A key issue in designing the placement module's memory allocator is internal fragmentation [391, 141]. Some fragmentation, which we call *classic fragmentation*, arises from the inability to pack variables onto pages (e.g., part of a page's space cannot store any variable). Another type of fragmentation, which we call *effective fragmentation*, arises from the page's lifetime exceeding some of the variables it stores; if even one wire on a page is alive, the entire page remains alive.

To reduce classic fragmentation, MAGE's placement stage uses techniques from slab allocators [74]. Each page contains only variables of a particular size. When a variable goes out of scope in the DSL, its "slot" in its page is marked as free. When a space for a variable must be allocated, MAGE's placement module looks for a free slot in a page containing variables of that size; if no such pages have free slots, it allocates a new page for variables of that size. The slab size is one

MAGE-virtual page. This ensures that no variable will straddle a page boundary. Just as in slab allocators, some leftover space at the end of a page may be unusable, but this can be controlled by tuning the page size. Unlike slab allocators, MAGE's placement module does not preserve object state across allocations.

To reduce effective fragmentation, MAGE's placement stage uses the following heuristic when allocating memory for a variable. If multiple pages, for the specified variable size, have free slots available, then MAGE uses the candidate page with the fewest free slots. This allows the number of live pages to decrease if the number of live variables decreases, by giving a chance for all variables on a page to die.

4.6.3 Second Stage: Replacement

We apply Belady's MIN algorithm [41]. MIN is theoretically optimal in the number of SWAP-IN operations, but it does not minimize the number of swap operations if SWAP-OUT operations are also considered. The reason is that only dirty pages need to be written back to storage (i.e., "swapped out"). Minimizing the number of swaps when taking this into account is NP-hard [166]. Regardless, MIN produces a solution with at most $2\times$ as many swaps as the theoretical optimum,[3] so it is useful in MAGE's replacement stage.

To use MIN, we first make a backward pass over the program to determine, each time a page is used, the time (instruction ID) at which it is used next. Then we make a forward pass over the program, using the annotated next use time to determine which page to swap out. This requires us to maintain a priority queue of resident pages, so that we can quickly identify which one's next use is farthest in the future. Each instruction, even if its arguments are already resident, requires us to also perform a `decrease_key` operation on the priority queue to adjust pages' next use time. Therefore, if N is the number of instructions and T is the number of pages that fit in memory, applying Belady's MIN algorithm is $\mathcal{O}(N \log T)$.

This stage outputs an instruction stream that contains swap directives and references wires by MAGE-physical address. To support this, MAGE's planner maintains a data structure that maps MAGE-virtual page numbers to MAGE-physical frame numbers, similar to a page table.

When planning a parallel/distributed program, the planner must be careful to not steal a page that is currently being used for network I/O. Thus, MAGE's replacement phase reads the network directives to infer the outstanding asynchronous network operations. When stealing pages, it issues *network barrier* directives, as necessary, to ensure that the engine waits for the relevant network I/Os to complete.

4.6.4 Third Stage: Scheduling

We introduce a parameter ℓ called the *lookahead*. To prefetch data, MAGE's scheduling algorithm attempts to move SWAP-IN directives ℓ instructions earlier in the instruction stream. However, this

[3] This occurs in the worst case where it evicts only dirty pages, but there is an optimal solution that evicts the same number of clean pages.

does not work if one of the ℓ intervening instructions uses the page frame into which we are bringing in data. We solve this by budgeting B extra physical page frames, called the *prefetch buffer*; the replacement stage is now run with a capacity of $T - B$ frames, not T frames. Data is brought asynchronously into a free slot in the prefetch buffer. Only when it is finally needed is it copied from the prefetch buffer into its destination physical page frame. Instead of SWAP-IN directives, the memory program contains ISSUE-SWAP-IN directives, which initiate the transfer of a page into memory, and FINISH-SWAP-IN directives, which block execution until a swap operation has completed. Ideally, swap operations will be scheduled such that FINISH-SWAP-IN never blocks, but it serves as an important fallback to prevent old/corrupt data from being used if the transfer is unpredictably delayed.

We use the prefetch buffer similarly to swap out pages. The page to be swapped out is copied into a free slot in the prefetch buffer and then swapped out to storage with an ISSUE-SWAP-OUT directive while execution of subsequent instructions continues. Unlike SWAP-IN operations, there is no clear deadline by which the write to storage must complete. Thus, we delay issuing a FINISH-SWAP-OUT directive for as long as possible; we only issue it when allocating a slot in the prefetch buffer fails. In such a situation, we identify the oldest ISSUE-SWAP-OUT operation, issue the FINISH-SWAP-OUT directive for it, and reclaim its page in the prefetch buffer.

One could eliminate the copying of pages to/from the prefetch buffer by rewriting future instructions. We did not implement this optimization because it would introduce additional complexity and MAGE performs well without it.

A natural question is how large B must be. SSDs have bandwidths less than 10 GB/s and latencies that are usually less than 1 ms. Based on these measurements, Little's Law gives: $B = 10\text{ GB/s} \cdot 1\text{ ms} = 10\text{ MB}$. For server-class machines, this is $< 1\%$ of physical memory. In practice, we use 16–32 MiB to account for burstiness/queuing, still only a small fraction of available memory. Thus, MAGE's scheduling promises to mask storage latency with only a small memory penalty.

4.7 Implementation

We implemented a prototype of MAGE in C++, including support for two protocols: garbled circuits and CKKS. Using `cloc`, we found that our implementation is $\approx 11,000$ lines of code, excluding comments and blank lines, broken down as follows: $\approx 2,800$ for common libraries used throughout MAGE (e.g., data buffering for I/O, configuration file parsing, etc.); $\approx 1,300$ for MAGE's planner; ≈ 900 for protocol drivers (not including the underlying cryptography); $\approx 1,000$ for MAGE's DSLs and libraries for those DSLs (e.g., for sharding data); $\approx 1,100$ for MAGE's engines; $\approx 1,600$ for SC programs written in MAGE's DSLs, used for testing and evaluating MAGE; $\approx 1,900$ for the underlying cryptography for garbled circuits, much of which is based on EMP-toolkit [473]; and ≈ 400 for in-progress (not yet complete) support for a third protocol. We build MAGE using `clang++` version 10.0.0 with the optimization flags `-Ofast -march=native`. MAGE runs as a Linux process, with no changes to kernel code.

4.7.1 Interpreter

Engine. The `Engine` class implements common functionality for the engine layer, including support for directives. It establishes pairwise TCP connections among workers within a single party, to support network directives. Swap directives are implemented using the `aio` facility provided by the kernel (not to be confused with POSIX `aio`); the swap file/device is `open`ed with the `O_DIRECT` flag. MAGE engines are implemented as class templates that extend (inherit from) the `Engine` class. The protocol driver class is provided to the engine as a template argument, so the engine can make calls to it. We avoided using virtual functions for this, as their overhead can be significant (e.g., for free XORs).

Protocol Driver. The protocol driver exposes the SC protocol's native operations to the engine as a set of methods. When the engine invokes these methods, it provides pointers to data to operate on, stored in a large array representing the MAGE-physical address space. The protocol driver specifies the type of entries in the engine's array, in effect dictating what each MAGE-physical address actually corresponds to for its protocol (plaintext bits, ciphertext bytes, etc.), and provides a plugin to the DSL so it can allocate MAGE-virtual memory accordingly. The protocol driver must not store pointers to dynamically allocated memory in the array. The reason is that the engine swaps out only the contents of the array, not including any dynamically-allocated memory it points to. In addition to the SC protocol's cryptographic routines, the driver manages all protocol-specific operations. This includes sharing protocol-specific state among workers within a party, obtaining input data, writing output data, and managing intra-party communication where necessary (e.g., sending garbled gates from the garbler to the evaluator).

4.7.2 Extending MAGE with New Protocols

To extend MAGE with a new protocol, one must, at minimum, write a protocol driver to support it. If the operations exposed by the new protocol driver are identical to those exposed by an existing protocol driver, then one can use the same engine that works with the existing protocol. Otherwise, one must implement a new engine or modify an existing engine. This involves deciding which instruction types the new engine will be compatible with. If the supported instruction types differ from what existing DSLs produce, then one may have to implement a new DSL or modify an existing DSL.

We implemented protocol drivers for garbled circuits and CKKS. Garbled circuits and CKKS support different operations, so we implemented a separate DSL (Integers vs. Batches) and engine (AND-XOR vs. Add-Multiply) for each protocol. This conveniently allows us to showcase MAGE's ability to support different implementations of each layer. That said, it is not uncommon for related SC protocols to expose similar interfaces. For example, the WRK protocol [474, 475] exposes the same interface as garbled circuits (AND-XOR), so support for WRK, if added, could reuse our Integer DSL and AND-XOR engine.

4.7.3 Garbled Circuit Protocol Driver

For garbled circuits, wires have uniform size, so we allow MAGE address spaces to be wire-addressed; the DSL is unaware of the size of wires in bytes. Some subcircuits used by the AND-XOR engine are based on those used by Obliv-C [503]. Our garbled circuit driver uses cryptographic kernels from EMP-toolkit [473]. We implement oblivious transfer (OT) using multiple background threads. Concurrently with our work, EMP-toolkit was updated to use the MiTCCRH hash function [206]; our implementation is based on an older version of EMP-toolkit based on fixed-key AES [45]. When we compare MAGE to EMP-toolkit in Section 4.8, we use the older version of EMP-toolkit so the comparison is fair. This is not a limitation of MAGE; our driver could be changed to use MiTCCRH.

4.7.4 CKKS Protocol Driver

CKKS ciphertexts vary in size depending on their level, so for CKKS' DSL and engine, MAGE address spaces are byte-addressed. The protocol driver provides a plugin to the DSL describing the particular wire sizes in bytes. It uses the CKKS implementation in Microsoft SEAL [410]. We chose parameters for CKKS that allow a multiplicative depth of 2. A challenge was that SEAL ciphertext objects contain pointers and dynamically-allocated memory. MAGE cannot swap such objects to storage (see Section 4.7.1). Thus, the protocol driver serializes ciphertexts using SEAL's built-in serialization methods when they are not in use; each operation (e.g., add, multiply) deserializes the arguments, computes the result, and then serializes the result. We quantify the cost of serialization in Section 4.8. This overhead is not fundamental; CKKS ciphertexts could be implemented as flat buffers, or homomorphic operations could be implemented to operate directly on serialized ciphertexts.

After a multiplication, CKKS ciphertexts are typically relinearized and rescaled before the next multiplication. But if two products are added (e.g., $ab + cd$), one can perform relinearization once for the overall result instead of for each multiplication separately (e.g., ab and cd). MAGE's DSL supports this optimization, which is crucial to achieve good performance on **rstats** and the linear algebra workloads.

4.8 Evaluation

In this section, we measure the performance impact of using MAGE.

4.8.1 Workloads

We now establish a set of SC workloads for our evaluation. Garbled circuits and CKKS support different operations—bitwise operations for garbled circuits, and add-multiply circuits of low multiplicative depth for CKKS—so we design separate workloads for each protocol. These workloads are data-intensive "kernels" that may be used as part of larger SC applications. We discuss larger SC applications in Section 4.8.8.

4.8.1.1 SMPC Collaborative Applications

One application of SMPC is federated data analytics [462, 378]. Aggregations (`GROUP BY` operations) and joins are particularly memory-intensive. A federated data analytics system may express equi-joins as set intersections (SI) and aggregations as set unions (SU), both of which can be implemented by merging sorted lists [378]. This inspires our first benchmark, **merge**: *merging sorted lists of records*. In some cases, the input lists may not be already sorted. This inspires our second benchmark, **sort**: *sorting a list of records*. For joins other than equi-joins, the system must fall back to a classic loop join. This is our third benchmark, **ljoin**: *loop join*. For concreteness, we assume that each record is 128 bits long, and that the first 32 bits are the key used for sorting or joining; the problem size n is the number of records per party.

Privacy-preserving machine learning inspires our fourth benchmark, **mvmul**: *matrix-vector multiply with 8-bit integers*. A proposal for secure neural network inference, XONN [397], suggests *binarizing* the neural network. This inspires our fifth benchmark, **binfclayer**: *binary fully-connected layer*. It consists of a series of XNOR and PopCount operations similar to multiplying a binary matrix by a binary vector, followed by a binary activation function. For simplicity, we do not include batch normalization.

4.8.1.2 CKKS Homomorphic Encryption

We restrict ourselves to workloads for which CKKS is efficient—workloads that can be expressed as arithmetic circuits of low multiplicative depth. The sixth workload is **rsum**: *sum of a list of real numbers*, which requires no multiplications. The seventh workload is **rstats**: *computing the mean and variance of real numbers*, which requires a multiplicative depth of 2. These represent simple data analytics workloads; the problem size n is the number of elements.

Our remaining workloads are inspired by machine learning and linear algebra. The eighth workload is **rmvmul**: *matrix-vector multiply with real numbers*. Finally, we consider two variants of matrix multiplication. The ninth workload is **n_rmatmul**: *matrix-matrix multiply with a naïve nested `for` loop*. The tenth workload is **t_rmatmul**: *tiled matrix-matrix multiply*. The problem size n is the length of one side of the matrix (also for **mvmul** and **binfclayer**).

4.8.1.3 Implementation of Workloads

For simplicity, our implementations of some of these workloads only support power-of-two sizes and power-of-two number of workers, but this is not a fundamental limitation of MAGE. Some workloads can, in principle, be optimized through streaming. For example, **rsum** could read each input one at a time, add the result to an accumulator, and then output the accumulator, instead of holding the entire input dataset in memory. We deliberately avoided such "optimizations," as they would not be possible if the workload were part of a larger computation whose intermediate results are held in memory. Thus, each workload operates in three non-overlapping phases: (1) the inputs are read into memory, (2) the computation is performed, materializing the output in memory, and (3) the output is written to a file.

For the parameters we chose, the CKKS scheme encrypts vectors of dimension 4096. Thus, each of our workloads for CKKS could be applied to 4096 instances of the problem in a SIMD fashion with no additional overhead. There are ways to use the 4096 slots in the vector to speed up a *single* problem, for example, by vectorizing matrix multiplication [256]. Our workloads, for simplicity, do not apply such techniques, but MAGE is not incompatible with them.

4.8.2 Empirical Methodology

We compare MAGE's performance to an upper bound and a lower bound. The upper bound, *OS Swapping*, is the performance when relying on the operating system's paging. The lower bound, *Unbounded*, is the performance when the entire computation fits in memory. We measure these three scenarios as follows.

1. *Unbounded.* MAGE's planner is run assuming enough memory to fit the program. Thus, MAGE's planner does not insert swap directives in the memory program. Finally, MAGE's engine executes the memory program outside of any `cgroup`.

2. *OS Swapping.* A memory program is generated in the same way as for the *Unbounded* solution. However, it is executed in a `cgroup` that limits physical memory to a fixed amount.

3. *MAGE.* MAGE's planner is run assuming a fixed physical memory capacity, minus the prefetch buffer and the interpreter's overhead. The resulting plan is run within a `cgroup` that limits physical memory to 1 GiB or 16 GiB, to ensure that the memory overhead fits in the limit.

All three scenarios execute the SC protocol using MAGE's interpreter. The benefit of this is that the same code executes SC in all three scenarios, giving us confidence that performance differences among the scenarios indeed stem from memory management behavior, not the implementation of SC. A downside, however, is that running MAGE's interpreter with OS Swapping may incur more page faults than an equivalent program that directly uses the corresponding cryptography library and incurs paging/swapping. In particular, when MAGE's interpreter allocates the array storing the program's data (Section 4.5), it prefaults it using the `MAP_POPULATE` flag. This interacts poorly with Linux paging when the array does not fit in memory—it forces pages in the array to be written out to swap space when the array is allocated and then faulted back in when they are later accessed. This should be kept in mind when interpreting our results.

Except where stated otherwise, we used `D16d_v4` instances on Microsoft Azure [344]. We chose this instance type for a few reasons. First, it has enough memory to fit the entire computation for most experiments, necessary for the Unbounded scenario. Second, it contains a local "temporary" SSD. We use it for swap space (one of its recommended uses [338]) and for the file containing the memory program. Third, it provides enough network bandwidth so as not to be a bottleneck for garbled circuits (we explore the WAN setting in Section 4.8.7).

We set MAGE's parameters as follows. For garbled circuits, we used a page size of 64 KiB, lookahead ℓ of 10,000 instructions, and prefetch buffer size B of 256 pages. For CKKS, we used a

page size of 2 MiB, lookahead ℓ of 100 instructions, and a prefetch buffer size B of 16 pages. Because CKKS ciphertexts are large, we used a larger page size (slab size) than for garbled circuits to reduce external fragmentation. Additionally, we left an additional 32–64 MiB of memory unused, to accommodate the memory used by MAGE's interpreter.

4.8.3 Comparison to Existing Frameworks

We compare MAGE's garbled circuits performance to that of EMP-toolkit. Our goal is to demonstrate that MAGE's techniques do not limit the performance of garbled circuits compared to an existing system. We use **merge** for the comparison. We implemented **merge** in EMP-toolkit's DSL, and used EMP-toolkit's library for merging sorted arrays.

We discovered that EMP-toolkit is an order of magnitude slower than MAGE. This was because EMP-toolkit performs a separate invocation of OT extension, which involves a network round-trip, each time an Integer input is read for the evaluator. Our garbled circuits implementation for MAGE does not have this problem because it performs OTs in larger batches using background threads, regardless of the units by which the program reads the input. To eliminate this effect, we exclude the time to read the input, for both EMP-toolkit and MAGE, for this experiment only; we measured the time to merge the two arrays once they are materialized in memory.

We also compare MAGE's CKKS performance on **rstats** to a C++ program that uses SEAL directly. The main source of overhead in MAGE is the need to deserialize the input ciphertexts and serialize the output ciphertext, for each instruction.

The results are shown in Figure 4.6 and Figure 4.7. The graphs on the left are zoomed in to smaller problem sizes to show the point where memory demand exceeds available physical memory. "OS" refers to scenario 2 in Section 4.8.2; "EMP" and "SEAL" refer to those systems similarly running in a `cgroup`. EMP performs about $3\times$ worse than OS when the problem fits in memory; when it does not, the relative overhead is small ($\approx 33\%$). We found that EMP performs worse than OS primarily due to (1) the overhead of its "real-time circuit optimization" feature, (2) inefficient data buffering when using the network, and (3) virtual function overhead when executing the circuit. OS uses MAGE's runtime, so it does not have these issues. SEAL is faster than OS when the problem fits in memory, but only slightly (less than 20%), indicating that the serialization overhead is not large. When the problem size does not fit in memory, SEAL improves further compared to OS but remains less than $2\times$ faster than OS. An explanation for why SEAL improves further compared to OS when the problem size does not fit in memory is the inefficiency of running MAGE's interpreter with OS Swapping (Section 4.8.2).

4.8.4 Overhead of Swapping Pages

We ran the three scenarios on all 10 workloads, using a 1 GiB memory limit. The results are shown in Figure 4.8. We ran 8 trials on different Azure instances (8 different pairs of instances, for garbled circuits) and plot the median; error bars are the quartiles. We additionally ran experiments using a 16 GiB memory limit. We increased the problem sizes so that their memory use exceeded 16 GiB (necessary for the OS scenario) but fit within the 64 GiB available on the virtual machines

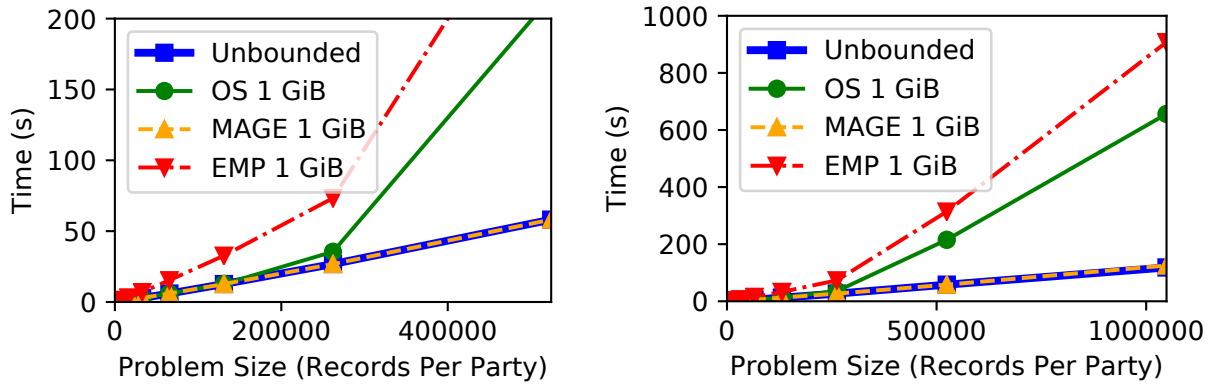


Figure 4.6: Comparison of MAGE and EMP-toolkit.

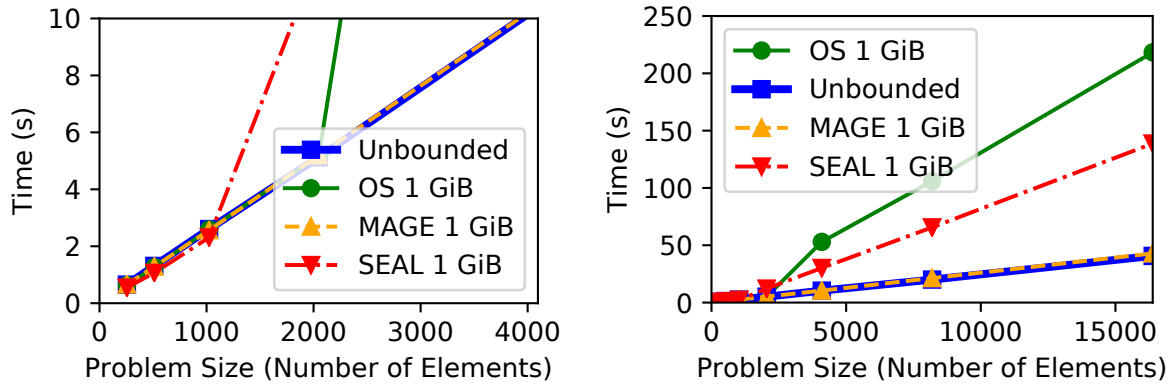


Figure 4.7: Comparison of MAGE and SEAL.

(necessary for the Unbounded scenario). Our methodology is the same as for the 1 GiB memory limit. We do not include **sort** in our results for the 16 GiB memory limit, because the intermediate bytecodes produced while planning were too large for the local SSD. The results are shown in Figure 4.9. MAGE outperforms OS swapping by at least 4× on 7 of the workloads, with improvements of ≈ 12× for **ljoin** and ≈ 10× for **rsum**. Its performance is within 15% of Unbounded for 7 of the workloads (including **sort** from Figure 4.8).

MAGE's improvement compared to OS is higher for **binfclayer** and **rmvmul** than for **mvmul**; although all three have similar access patterns, **mvmul** has lower memory intensity because multiplying integers in a garbled circuit has high overhead. For complex access patterns, like **merge** and **sort**, MAGE's improvement is not markedly higher than for simple scans like **ljoin**, **rsum**, and **rstats** (note that both input tables for **ljoin** fit in memory; it is the *output*, populated in order, that does not fit). MAGE is less affected by high memory intensity than OS, allowing it to perform well.

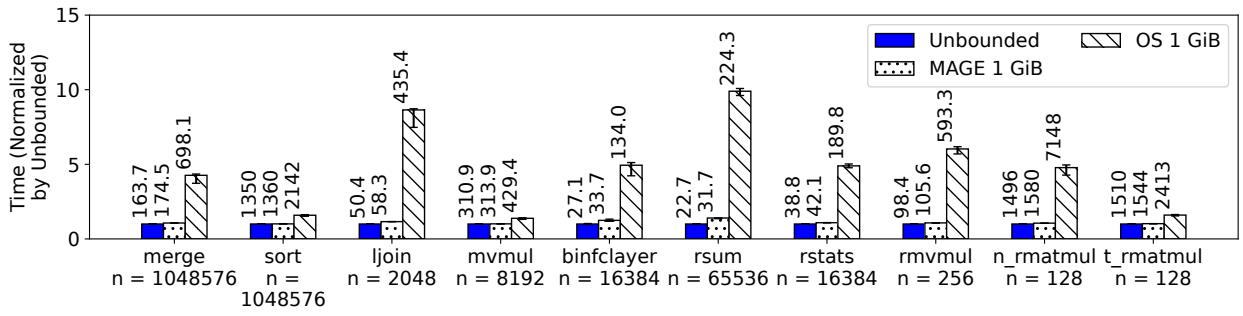


Figure 4.8: Performance of Unbounded, OS Swapping, and MAGE, normalized by the time for Unbounded; absolute times, in seconds, are printed at the upper left corner of each bar.

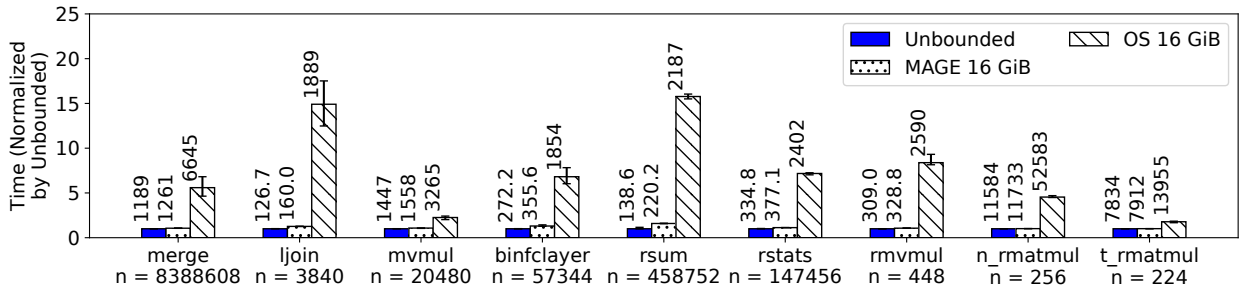


Figure 4.9: Repeat of Figure 4.8, with larger problem sizes and a 16 GiB memory limit (note the larger y-axis scale).

4.8.5 Overhead of Planning

The time and peak memory use for planning each workload for the MAGE scenario in Figure 4.8 and Figure 4.9 is shown in Table 4.1. These measurements were collected by running `/usr/bin/time -v` when invoking MAGE's planner. Note that MAGE's planning is outside of the critical path: for a given circuit, MAGE's planner can be run before the parties' inputs are known. For garbled circuits, although the garbled circuit $\widetilde{C}$ cannot be reused if the computation is re-run, MAGE's memory program *can* be safely reused.

The planning time and final memory program size are linear in the size of the *computation* (size of C), not in the size of the memory demand. Nevertheless, the planning times are generally less than the time to perform the execution and the planner's memory consumption is significantly smaller than the available memory at runtime for all experiments.

Generating memory programs for CKKS is more efficient than for garbled circuits. This is because each instruction for CKKS operates on more memory than for garbled circuits, which means that the problem sizes that fill a given physical memory size tend to require smaller bytecodes for CKKS than for garbled circuits. For example, an instruction operating on integers in a garbled circuit program may operate on a few kilobytes of memory (each bit of each integer is 16 bytes), but

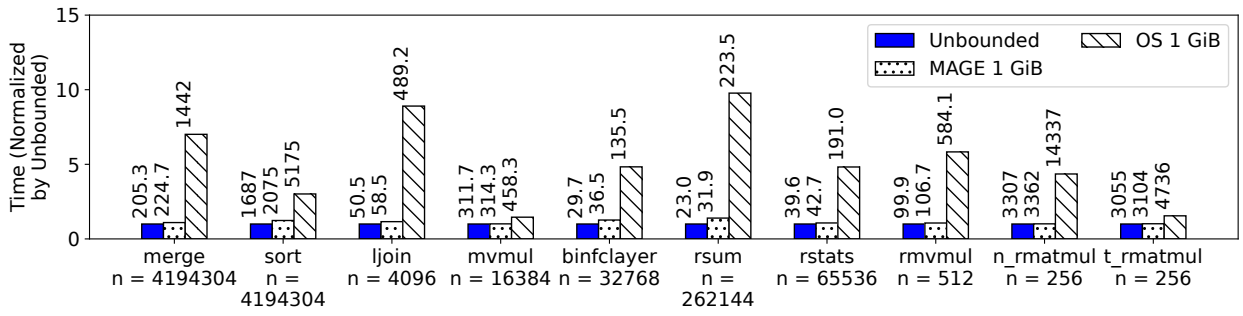


Figure 4.10: Normalized performance of Unbounded, OS Swapping, and MAGE, parallelized over $p = 4$ workers (per party).

Problem	Time, Figure 4.8	Mem., Figure 4.8	Time, Figure 4.9	Mem., Figure 4.9
merge	38.0	42.6	291.6	299.4
sort	367.3	42.7	N/A	N/A
ljoin	6.7	121.0	23.6	411.4
mvmul	56.0	527.5	298.2	3268
binfclayer	77.2	19.1	1041	165.7
rsum	0.04	9.6	0.29	30.2
rstats	0.04	10.9	0.34	48.5
rmvmul	0.09	16.4	0.24	36.9
n_rmatmul	2.2	246.1	18.6	1927
t_rmatmul	2.3	246.5	12.9	1246

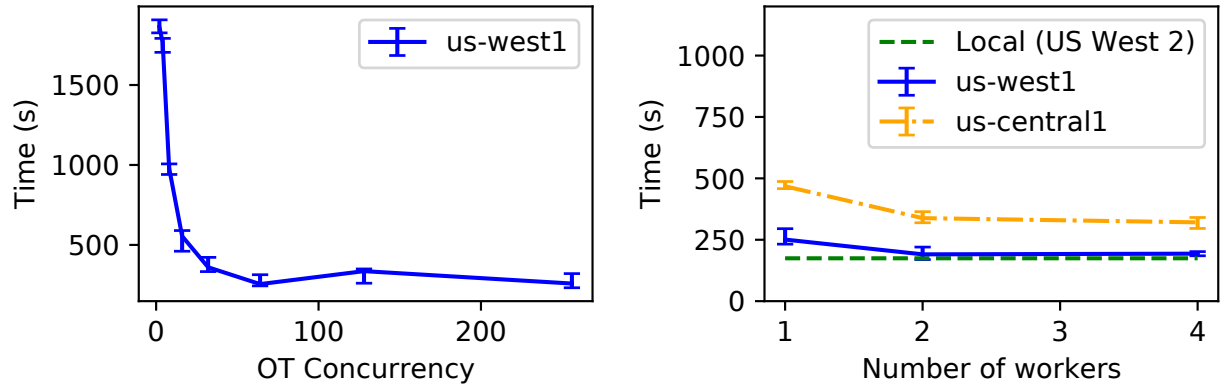
Table 4.1: Planning times (s) and peak memory use of the planner (MiB) for workloads in Figure 4.8 and Figure 4.9.

for CKKS, each instruction operates on a *vector* of real numbers, whose encrypted size is hundreds of kilobytes.

For CKKS, the final memory programs were < 100 MiB for Figure 4.8 and < 1 GiB for Figure 4.9. For garbled circuits other than **sort**, they were < 5 GiB for Figure 4.8 and < 65 GiB for Figure 4.9. For **sort**, it was less than < 25 GiB for Figure 4.8. MAGE's planner requires about 4–5$\times$ times more storage space than the final memory program due to the need to materialize intermediate bytecodes of similar size, but this could be optimized by pipelining stages of MAGE's planner where it is possible to do so (e.g., replacement and scheduling in Figure 4.4).

4.8.6 Impact of Parallelism

We now explore how the relative performance of Unbounded, OS, and MAGE are affected by parallelizing the computation. We did experiments parallelizing the computation across four workers



(a) Time to run **merge** vs. number of concurrent OTs.

(b) Time to run **merge** vs. number of workers.

Figure 4.11: Wide-area garbled circuit performance in MAGE.

(per party, for garbled circuits). We place each worker on a separate VM instance, each with a separate SSD.

We ran each experiment three times, using the same cluster of machines for all trials, and report the median in Figure 4.10. Most experiments follow a similar pattern as Figure 4.8, indicating that MAGE's performance gains persist when we parallelize the computation. For two experiments, **merge** and **sort**, MAGE's improvement over OS Swapping visibly increases. Whereas the other workloads are parallelized by splitting the input among the workers in a communication phase at the beginning and then computing independently thereafter, **merge** and **sort** have a communication phase in the *middle* of the computation (several such phases in the case of **sort**). That OS Swapping performs worse for these workloads, but MAGE does not, suggests that the OS virtual memory system might be introducing jitter, which interacts poorly with the communication phase and induces stragglers.

4.8.7 SMPC in Wide-Area Networks

SC does not always require significant data transfer over the wide area. In HE, computation is done by a single logical party. Even in SMPC, there may be ways for multiple parties to co-locate for an SMPC computation while remaining physically and logically distinct. But in some cases, it is desirable to run SMPC over a wide-area network. We explore this below.

We measure performance of garbled circuits with the two parties hosted on different cloud providers. The garbler was always on Azure in West US 2 (Washington). The evaluator was on Google Cloud (`n2-highcpu-2` [199]). We compare two setups: one where the evaluator was in `us-west1` (Oregon) and one where it was in `us-central1` (Iowa).

Initially, higher latencies and limited single-flow bandwidth limited performance. For example, the round-trip time in the Oregon setup was ≈11 ms, which made OTs a bottleneck.

First, we tuned the local TCP stack, increasing the maximum window size to 32 MiB. Then, we increased the number of OT rounds performed concurrently, pipelining multiple OT rounds over a single connection, which significantly improved performance (Figure 4.11a). Additionally, we explore parallelizing the computation, assigning multiple workers to the same machine, so that multiple TCP flows are used. The results are in Figure 4.11b. The dashed line at the bottom is the time to run the experiment with both the garbler and evaluator on Azure (taken from Figure 4.8). For the Oregon setup, we can come close to the Local performance using two flows. The Iowa setup is more challenging because less bandwidth is available per flow. Using multiple parallel flows helps, but the performance improvement in the Iowa setup is limited by variation in wide-area flow performance, which induces stragglers.

In both cases, the performance overhead of operating in the wide area is less than the performance overhead of swapping (Figure 4.8), indicating that MAGE's techniques confer substantial benefit even in wide-area settings.

4.8.8 Applications

For these experiments, we did not use `cgroups` to limit RAM. The OS and MAGE setups ran using all of the available RAM.

4.8.8.1 Detecting Password Reuse

When users reuse a password across multiple websites, they become prone to "credential stuffing" attacks, in which an attacker uses a user's password leaked by one site to compromise that user's account on other sites. To address this problem, sites may wish to identify which of their users reuse their passwords on other sites [472]. Senate [378, Query 2 in Section 2] proposes a protocol for this. First, the sites arrange to assign user IDs and hash passwords such that they will match *across* sites. Then, they use SMPC to detect which user IDs are shared between the sites and have the same password hash. Note that user IDs and password hashes cannot be shared directly, since they are sensitive (the hashes can be reversed).

We write a two-party version of the password reuse program in MAGE's DSL for garbled circuits, based on Senate's password reuse program. Senate uses a different SMPC protocol, so its results are not directly comparable to ours.

We use MAGE to scale the password reuse program to 2^{27} users per party, which requires 1.125 TiB on each party. A single `D16d_v4` instance does not have enough swap space. Thus, we use four `D16d_v4` instances on Azure for the garbler party, and four `n2-highmem-4` instances on Google Cloud [199] for the evaluator party. As explored in Section 4.8.7, we use two workers per instance (total of eight workers per party) to efficiently use wide-area network bandwidth. The results are shown in Figure 4.12. For a given time budget, MAGE increases the number of user-password records by $\approx 3\times$. This improvement may have been larger had we been able to obtain `Ddv4`-series instances with a greater swap-space-to-RAM ratio.

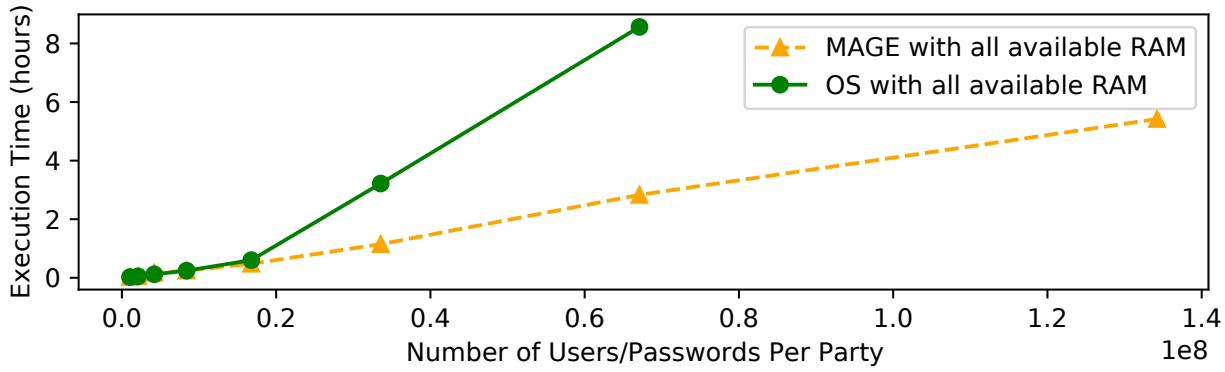


Figure 4.12: Scaling password reuse detection with MAGE.

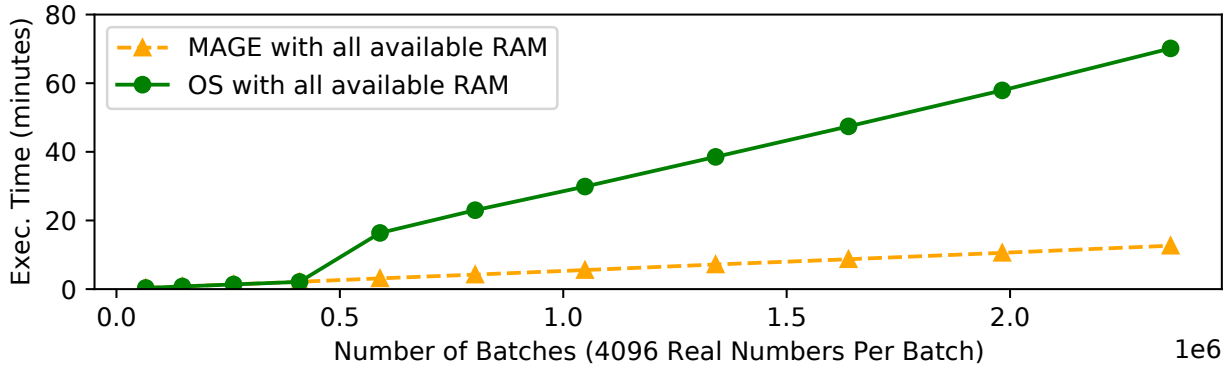


Figure 4.13: Scaling computational PIR with MAGE.

4.8.8.2 Private Information Retrieval

Private Information Retrieval (PIR) is a family of protocols that allow a user to retrieve a data item at a particular index from a database without the database learning which item was accessed. PIR can be used to support private queries on public data [469]. We evaluate MAGE by using CKKS to instantiate the classic Kushilevitz-Ostrovsky single-server computational PIR scheme [300, Section 3]. PIR's access pattern is particularly simple—a linear scan over the database—so ad-hoc approaches to prefetching, or multi-threading to improve swap performance, may be quite effective. Our focus is on what MAGE optimizes *automatically*, so we do not include such ad-hoc optimizations in the OS baseline. We use a single worker (thread) to compute the PIR. The database consisted of plaintext data pre-encoded into batches to use with CKKS. We wrote a DSL program that populates the database (with hardcoded elements) and then performs a PIR query on it; the reported measurements are the time to perform the PIR query, not including the time to populate the database. The results are in Figure 4.13. For a given time budget, MAGE allows for $\approx 5\times$ as

many database elements to be processed.

4.9 Related Work

Much existing work has looked at high-performance algorithms for SMPC [475, 133, 132, 266, 267] and HE [116, 184]. These works focus on the cryptography, not how to manage a computer's resources to perform large computations efficiently.

A complementary line of work explores tailoring SMPC computations to a specific application [260, 515, 397, 109]. The goal of MAGE is to perform the same computation more efficiently, so its techniques generalize across different applications. For an application, one may first simplify the computation using application-specific observations, and then execute the resulting computation as efficiently as possible.

Research works including Fairplay [332], HEKM [230], KSS [288], MLB [349], PCF [287], and TinyGarble [436] are frameworks for garbled circuit execution. We described many of them in Section 4.2.4. One work [90] explores parallelizing execution of a garbled circuit, using programming language tools to automatically extract parallelism. None of them explore how to efficiently swap memory to storage, as MAGE does.

There already exist many DSLs and compilers for SMPC [221, 473, 321, 503, 348, 214, 514] and HE [99, 134, 459]. These tools often aim to make SC more accessible to non-expert developers, by automatically optimizing the SC program. MAGE addresses the complementary problem of executing the resulting SC circuit more efficiently. To use an existing tool with MAGE (as in Figure 4.2), one could modify it to output its optimized circuits in one of MAGE's DSLs, and then run MAGE's planner on that DSL code. Alternatively, one could modify the tool to output a bytecode directly usable by MAGE's planner (e.g., the "Virtual Bytecode" in Figure 4.4).

AIFM [402] uses similar C++ language features as MAGE's DSLs. AIFM uses them at runtime for fine-grained memory management. In contrast, MAGE (1) executes DSL programs only to extract the memory access pattern during the planning phase and (2) manages memory at the granularity of pages.

There is an extensive literature concerning memory management in traditional operating systems [41, 140, 42, 141, 142]. A related line of work looks at how operating systems can give memory-intensive applications, such as scientific simulations, more control over paging [212]. While these works focus primarily on paging in the classic sense, our work explores memory programming. Additionally, our work, unlike scientific simulations, is capable of *general* computations within SC. Scheduling page movement according to real-time constraints imposed by computation also draws from the real-time scheduling literature [320]. These techniques do not manage memory directly and are complementary to ours.

Some systems in other domains, like neural network training, formulate memory management problems as an integer linear program and use an exponential-time solver [253]. This approach exploits the high-level structure of the application to coarsen the dataflow graph. For MAGE, the dataflow graph is much larger because *general* SC computations do not conform to any particular high-level structure. By operating on a program representation of the circuit (Section 4.4.2),

MAGE does coarsen the graph, but it nevertheless remains enormous. Thus, we use our staged approach (Section 4.6) to find a good approximation.

Some systems use compiler-generated hints or observations of past memory accesses or past working sets (e.g., from prior invocations of a program) to perform targeted prefetching [350, 511, 215, 213, 335, 457] and approximate Belady's algorithm (MIN) [435]. SC's obliviousness and our memory programming approach allow MAGE to compute the memory access pattern for the full program without first running it and apply these techniques by using the memory access pattern directly.

The recent DEMAND-MIN [252] algorithm combines MIN with prefetching. DEMAND-MIN tells which item to evict given an access pattern sequence and prefetch sequence fixed in advance. It is not directly applicable to MAGE because MAGE's prefetch sequence is not fixed in advance.

At a technical level, MAGE's planning is similar to register allocation in compiler theory [102, 125, 452, 483]—variables, registers, and memory in register allocation correspond to wire values, slots in memory, and storage swap space in the context of MAGE. The key difference is that register allocators must deal with conditional branches whose outcomes cannot be predicted at compile time. From the perspective of register allocation, the entire circuit that MAGE operates on would be viewed as a single basic block. We discussed a result from register allocation theory for a single basic block in Section 4.6.3. Another result is that, for a *fixed* number of registers, there is a linear-time algorithm that can reorder instructions within a structured program to optimize its register allocation [60, Section 3.2] (though the time is exponential in the number of registers).

4.10 Conclusion

This chapter explores how to efficiently execute SC computations that do not fit in memory. Our key observation is that SC is inherently oblivious. This enables memory programming, in which one computes the access pattern of an SC program in advance and uses it to produce a memory management plan. Memory programming is an application of the technique in Section 3.1.1—it allows for memory management according to the structure of the computation, namely its obliviousness. In applying this technique, MAGE runs SC up to an order of magnitude faster than the OS virtual memory system and can execute some SC programs that do not fit in memory at nearly in-memory speeds. This demonstrates the potential of the technique in Section 3.1.1 to make expressive cryptography, such as SMPC and FHE, practical for more applications.

Chapter 5

Supporting Cryptography in Low-Power Wireless Systems with Performant TCP

This is the second of two chapters exploring the techniques in Section 3.1. We focus in this chapter on how to provide networking support for cryptography for applications that run on low-power, resource constrained, wireless embedded devices. Such devices have been used in scientific applications, such as environmental monitoring and structural monitoring, but also in the Internet of Things (IoT) space, where data is sensitive in nature and expressive cryptography is particularly relevant. For example, the Thread Group [450] is an industry consortium in the home automation space formed around interoperability of low-power wireless technology based on IEEE 802.15.4. As described in Section 2.2.1.3, expressive cryptography can have significant networking costs, for example, due to the need to transfer large ciphertexts among devices. Unfortunately, supporting expressive cryptography for resource-constrained wireless embedded devices is particularly challenging. The reason is that the standard network protocols for reliable, high-throughput data delivery, like TCP, used to support expressive cryptography in regular networks, are seen as unsuitable for ultra low-power wireless networks like those based on IEEE 802.15.4.

This chapter studies TCP for resource-constrained wireless embedded devices, with particular attention to the interaction between the transport layer and the link layer. We apply the technique in Section 3.1.2, generically improving the underlying network to make it easier to support applications built on expressive cryptography. We identify subtle but important modifications to both, achieving TCP goodput within 25% of an upper bound (5–40% higher than prior results) and low-power operation commensurate to CoAP, a simpler alternative to TCP used in these networks. In doing so, we show that TCP can be made viable and performant in networks based on IEEE 802.15.4, making it easier to meet the networking demands of expressive cryptography in this space. That said, as we explain in Section 5.3, performant TCP over IEEE 802.15.4 broadly benefits the Internet of Things beyond just making it easier to run expressive cryptography, as is expected for systems based on the technique in Section 3.1.2.

We aim to explain our study and its conclusions in the broad context of the low-power wireless space, without restricting our attention to only its implications for expressive cryptography. To that end, this chapter provides background and context on low-power wireless networks and describes

and evaluates our proposed system, *TCPlp*, independently of expressive cryptography. In doing so, we hope to bring out the full value and generality of our conclusions and contributions.

5.1 Introduction

Research on wireless networks of low-power, resource-constrained, embedded devices—in IETF terms, low-power and lossy networks (LLNs) [458]—blossomed in the late 1990s. To obtain freedom to tackle the unique challenges of LLNs, researchers initially departed from the established conventions of the Internet architecture [163, 220]. As the field matured, however, researchers found ways to address these challenges *within* the Internet architecture [233]. Since then, it has become commonplace to use IPv6 in LLNs via the 6LoWPAN [347] adaptation layer. IPv6-based routing protocols, like RPL [487], and application-layer transport protocols over UDP, like CoAP [100], have become standards in LLNs. Most wireless sensor network (WSN) operating systems, such as TinyOS [220, 309], RIOT [29], and Contiki [153], ship with IP implementations enabled and configured. Major industry vendors offer branded and supported 6LoWPAN stacks (e.g., TI SimpleLink, Atmel SmartConnect). A consortium, Thread [450], has formed around 6LoWPAN-based interoperability.

Despite these developments, transport in LLNs has remained ad-hoc and TCP has received little serious consideration. Many embedded IP stacks (e.g., OpenThread [360]) do not even support TCP, and those that do support TCP implement only a subset of its features (Section 5.5.1). The conventional wisdom is that IP holds merit, but *TCP is ill-suited to LLNs*. This view is represented by concerns about TCP such as the following.

- "TCP is not light weight ... and may not be suitable for implementation in low-cost sensor nodes with limited processing, memory and energy resources." [368] (Similar argument in [150], [247].)
- That "TCP is a connection-oriented protocol" is a poor match for WSNs, "where actual data might be only in the order of a few bytes." [388] (Similar argument in [368].)
- "TCP uses a single packet drop to infer that the network is congested." This "can result in extremely poor transport performance because wireless links tend to exhibit relatively high packet loss rates." [366] (Similar argument in [151], [152], [247].)

Such viewpoints have led to a plethora of WSN-specialized protocols and systems [368, 393, 467] for reliable data transport, such as PSFQ [465], STCP [247], RCRT [366], Flush [279], RMST [438], Wisden [492], CRRT [5], and CoAP [77], and for unreliable data transport, like CODA [466], ESRT [407], Fusion [234], CentRoute [439], Surge [308], and RBC [510].

As LLNs become part of the emerging Internet of Things (IoT), it behooves us to re-examine the transport question, with attention to how the landscape has shifted: (1) As part of IoT, LLNs must be interoperable with traditional TCP/IP networks; to this end, using TCP in LLNs simplifies IoT gateway design. (2) Popular IoT application protocols, like MQTT [352] and ZeroMQ [507],

Challenge	Technique	Observed Improvement
Resource Constraints	Zero-Copy Send	Send Buffer: 50% less memory
	In-Place Reassembly	Receive Buffer: 38% less memory
Link-Layer Properties	Large Maximum Segment Size	TCP Goodput: 4–5× higher
	Link Retry Delay	TCP Segment Loss: 6% → 1%
Energy Constraints	Adaptive Duty Cycle	HTTP Latency: ≈ 2× lower
	Link-Layer Queue Management	TCP Radio Duty Cycle: 3% → 2%

Table 5.1: Impact of techniques to run full-scale TCP in LLNs.

assume that TCP is used at the transport layer. (3) Some IoT application scenarios demand high link utilization and reliability on low-bandwidth lossy links. Embedded hardware has also evolved substantially, prompting us to revisit TCP's overhead. In this context, **this chapter seeks to determine: Do the "common wisdom" concerns about TCP hold in a modern IEEE 802.15.4-based LLN? Is TCP (still) unsuitable for use in LLNs?**

To answer this question, we leverage the fully-featured TCP implementation in the FreeBSD Operating System (rather than a limited locally-developed implementation) and refactor it to work with the Berkeley Low-Power IP Stack (BLIP), Generic Network Stack (GNRC), and OpenThread network stack, on two modern LLN platforms (Section 5.5). Naïvely running TCP in an LLN indeed results in poor performance. However, upon close examination, we discover that this is not caused by the expected reasons, such as those listed above. The *actual* reasons for poor TCP performance include (1) small link-layer frames that increase TCP header overhead, (2) hidden terminal effects over multiple wireless hops, and (3) poor interaction between TCP and a duty-cycled link. Through a systematic study of TCP in LLNs, we develop techniques to resolve these issues (Table 5.1), uncover why the generally assumed problems do not apply to TCP in LLNs, and show that TCP performs well in LLNs once these issues are resolved:

We find that **full-scale TCP fits well within the CPU and memory constraints of modern LLN platforms** (Section 5.5, Section 5.6). Owing to the low bandwidth of a low-power wireless link, a small window size (≈ 2 KiB) is sufficient to fill the bandwidth-delay product and achieve good TCP performance. This translates into small send/receive buffers that fit comfortably within the memory of modern WSN hardware. Furthermore, we propose using an atypical Maximum Segment Size (MSS) to manage header overhead and packet fragmentation. As a result, **full-scale TCP operates well in LLNs, with 5–40 times higher throughput than existing (relatively simplistic) embedded TCP stacks** (Section 5.6).

Hidden terminals are a serious problem when running TCP over multiple wireless hops. We propose adding a delay d between link-layer retransmissions, and demonstrate that it effectively reduces hidden-terminal-induced packet loss for TCP. We find that, because a small window size is sufficient for good performance in LLNs, **TCP is quite resilient to spurious packet losses, as the congestion window can recover to a full window quickly after loss** (Section 5.7).

To run TCP in a low-power context, we *adaptively* duty-cycle the radio to avoid poor interac-

tions with TCP's self-clocking behavior. We also propose careful link-layer queue management to make TCP more robust to interference. We demonstrate that **TCP can operate at low power, comparable to alternatives tailored specifically for WSNs**, and that **TCP brings value for real IoT sensor applications** (Section 5.8).

We conclude that TCP is entirely capable of running on IEEE 802.15.4 networks and low-cost embedded devices in LLN application scenarios (Section 5.9). Since our improvements to TCP and the link layer maintain seamless interoperability with other TCP/IP networks, we believe that a TCP-based transport architecture for LLNs could yield considerable benefit.

In summary, this chapter's contributions are:

1. We implement a full-scale TCP stack for low-power embedded devices and reduce its resource usage.
2. We identify the actual issues causing poor TCP performance and develop techniques to address them.
3. We explain why the expected insurmountable reasons for poor TCP performance actually do not apply.
4. We demonstrate that, once these issues are resolved, TCP performs comparably to LoWPAN-specialized protocols.

Table 5.1 lists our techniques to run TCP in an LLN. Although prior LLN work has already used various forms of link-layer delays [488] and adaptive duty-cycling [499], our work shows, where applicable, how to adapt these techniques to work well with TCP, and demonstrates that they can address the challenges of LLNs within a *TCP-based* transport architecture.

5.2 Background and Related Work

Since the introduction of TCP, a vast literature has emerged, focusing on improving it as the Internet evolved. Some representative areas include congestion control [249, 164, 204, 3], performance on wireless links [36, 10], performance in high-bandwidth environments [76, 174, 257, 207, 6], mobility [432], and multipath operation [389]. Below, we discuss TCP in the context of LLNs and embedded devices.

5.2.1 Low-Power and Lossy Networks (LLNs)

Although the term *LLN* can be applied to a variety of technologies, including LoRa and Bluetooth Low Energy, we restrict our attention in this chapter to **embedded networks using IEEE 802.15.4**. Such networks are called LoWPANs [299]—Low-Power Wireless Personal Area Networks—in contrast to WANs, LANs (802.3), and WLANs (802.11). Outside of LoWPANs, TCP has been successfully adapted to a variety of networks, including serial [248], Wi-Fi [36], cellular [34, 240], and satellite [34, 10] links. While an 802.15.4 radio can in principle be added as a NIC to any

device, we consider only *embedded* devices where it is the primary means of communication, running operating systems like TinyOS [220], RIOT [29], Contiki [153], or FreeRTOS. These devices are currently built around microcontrollers with Cortex-M CPUs, which lack MMUs. Below, we explain how LoWPANs are different from other networks where TCP has been successfully adapted.

5.2.1.1 Resource Constraints

When TCP was adopted by ARPANET in the early 1980s, Internet citizens—typically minicomputers and high-end workstations, but not yet personal computers—usually had at least 1 MiB of RAM. 1 MiB is tiny by today's standards, yet the LLN-class devices we consider in this work have *1-2 orders of magnitude less RAM than even the earliest computers connected with TCP/IP*. Due to energy constraints, particularly SRAM leakage, RAM size in low-power MCUs does not follow Moore's Law. For example, comparing Hamilton [274], which we use in this work, to TelosB [380], an LLN platform from 2004, shows only a 3.2× increase in RAM size over 16 years. This has caused LLN-class embedded devices to have a different balance of resources than conventional systems, a trend that is likely to continue well into the future. For example, whereas conventional computers have historically had roughly 1 MiB of RAM for every MIPS of CPU, as captured by the 3M rule, Hamilton has ≈ 50 DMIPS of CPU but only 32 KiB of RAM.

5.2.1.2 Link-Layer Properties

IEEE 802.15.4 is a low-bandwidth, wireless link with an MTU of only 104 bytes. The research community has explored using TCP with links that are *separately* low-bandwidth, wireless [36], or low-MTU [248], but addressing these issues *together* raises new challenges. For example, RTS-CTS, used in WLANs to avoid hidden terminals, has high overhead in LoWPANs [488, 234] due to the small MTU—control frames are comparable in size to data frames. Thus, LoWPAN researchers have moved away from RTS-CTS, instead carefully designing application traffic patterns to avoid hidden terminals [279, 379, 234]. Unlike Wi-Fi/LTE, LoWPANs do not use physical-layer techniques like adaptive modulation/coding or multi-antenna beamforming. Thus, they are directly impacted by link quality degradation due to varying environmental conditions [443, 379]. Additionally, IEEE 802.15.4 coexists with Wi-Fi in the 2.4 GHz frequency band, making Wi-Fi interference particularly relevant in indoor settings [317]. As LoWPANs are *embedded* networks, there is no human in the loop to react to and repair bad link quality.

5.2.1.3 Energy Constraints

Embedded nodes—the "hosts" of an LLN—are subject to strict power constraints. Low-power radios consume almost as much energy listening for a packet as they do when actually sending or receiving [274, 324]. Therefore, it is customary to *duty-cycle* the radio, keeping it in a low-power sleep state, in which it cannot send or receive data, most of the time [498, 379, 233]. The radio is only *occasionally* turned on to send/receive packets or determine if reception is likely. This requires *Media Management Control (MMC)* protocols [498, 379, 233] at the link layer to ensure

that frames destined for a node are delivered to it only when its radio is on and listening. Similarly, the CPU also consumes a significant amount of energy [274], and must be kept idle most of the time.

Over the past 20 years, LLN researchers have addressed these challenges, but only in the context of special-purpose networks highly tailored to the particular application task at hand. The remaining open question is how to do so with a general-purpose reliable transport protocol like TCP.

5.2.2 TCP/IP for Embedded LLN-Class Devices

In the late 1990s and early 2000s, developers attempted to bring TCP/IP to embedded and resource-constrained systems to connect them to the Internet, usually over serial or Ethernet. Such systems [78, 117] were often designed with a specific application—often, a web server—in mind. These TCP/IP stacks were tailored to the specific applications at hand and were not suitable for general use. uIP ("micro IP") [150], introduced in 2002, was a standalone *general* TCP/IP stack optimized for 8-bit microcontrollers and serial or Ethernet links. To minimize resource consumption to run on such platforms, uIP omits standard features of TCP; for example, it allows only a single outstanding (unACKed) TCP segment per connection, rather than a sliding window of in-flight data.

Since the introduction of uIP, embedded networks have changed substantially. With *wireless* sensor networks and IEEE 802.15.4, various low-power networking protocols have been developed to overcome lossy links with strict energy and resource constraints, from S-MAC [498], B-MAC [379], X-MAC [91], and A-MAC [158], to Trickle [310] and CTP [189]. Researchers have viewed TCP as unsuitable, however, questioning end-to-end recovery, loss-triggered congestion control, and bi-directional data flow in LLNs [152]. Furthermore, WSNs of this era typically did not even use IP; instead, each WSN was designed specifically to support a particular application [330, 492, 280]. Those that require global connectivity rely on application-specific "base stations" or "gateways" connected to a TCP/IP network, treating the LLN like a peripheral interconnect (e.g., USB, bluetooth) rather than a network in its own right. This is because the prevailing sentiment at the time was that LLNs are too different from other types of networks and have to operate in too extreme conditions for the layered Internet architecture to be appropriate [163].

In 2007, the 6LoWPAN adaptation layer [347] was introduced, enabling IPv6 over IEEE 802.15.4. IPv6 has since been adopted in LLNs, bringing forth IoT [233]. uIP has been ported to LLNs [156], and IPv6 routing protocols, like RPL [487], and UDP-based application-layer transports, like CoAP [100], have emerged in LLNs. Representative operating systems, like TinyOS and Contiki, implement UDP/RPL/IPv6/6LoWPAN network stacks with IEEE 802.15.4-compatible MMC protocols for 16-bit platforms like TelosB [380].

TCP, however, is not widely adopted in LLNs. The few LLN studies that use TCP [155, 219, 233, 238, 277, 513, 195] generally use a simplified TCP stack (Section 5.5.1), such as uIP.

In summary, despite the acceptance of IPv6, LLNs remain highly tailored at the transport layer to the application at hand. They typically use application-specific protocols on top of UDP; of such protocols, CoAP [77] has the widest adoption. In this context, this chapter explores whether

adopting TCP—and more broadly, the ecosystem of IP-based protocols, rather than IP alone—might bring value to LLNs moving forward.

5.3 Motivation

We first describe the benefits of TCP in LLNs in general terms and then describe anemometry, a candidate application of TCP in LLNs.

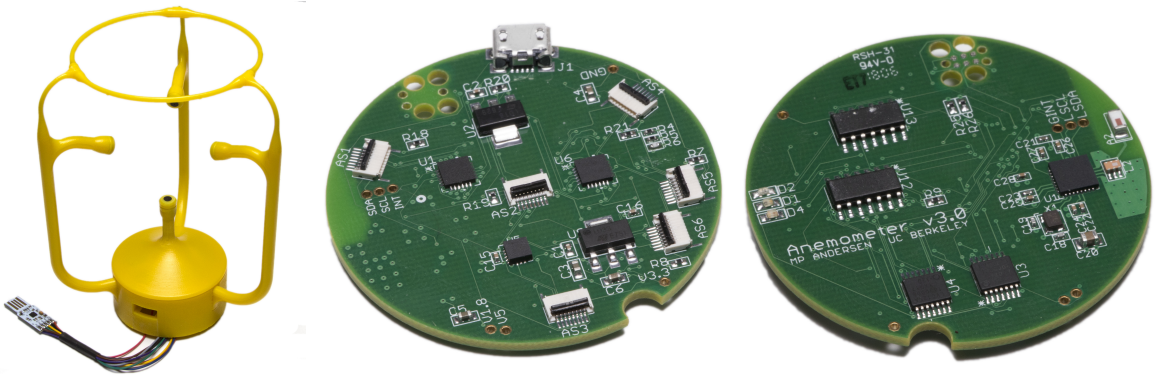
5.3.1 The Case for TCP in LLNs

As explained in Section 5.2, LLN design has historically been highly tailored to the specific application task at hand, for maximum efficiency. For example, PSFQ broadcasts data from a single source node to all others, RMST supports "directed diffusion" [242], and CoAP is tied to REST semantics. But embedded networks are not just isolated devices (e.g., peripheral interconnects like USB or Bluetooth)—they are now true Internet citizens, and should be designed as such.

In particular, the recent megatrend of IoT requires LLNs to have a greater degree of *interoperability* with regular TCP/IP networks. Yet, LLN-specific protocols lack a clear separation between the transport and application layers, requiring *application-layer gateways* to communicate with TCP/IP-based services. This has encouraged IoT applications to develop as vertically-integrated silos, where devices cooperate only within an individual application or a particular manufacturer's ecosystem, with little to no interoperability *between* applications or with the general TCP/IP-based Internet. This phenomenon, sometimes called the "CompuServe of Things," is a serious obstacle to the IoT vision [502, 311, 180, 339, 484]. In contrast, other networks are seamlessly interoperable with the rest of the Internet. Accessing a new web application from a laptop does not require any new functionality at the Wi-Fi access point, but running a new application in a gateway-based LLN *does* require additional application-specific functionality to be installed at the gateway.

In this context, TCP-enabled LLN devices would be first-class citizens of the Internet, natively interoperable with the rest of the Internet via TCP/IP. They could use IoT protocols that assume a TCP-based transport layer (e.g., MQTT [352]) and security tools for TCP/IP networks (e.g., stateful firewalls), without an application-layer gateway. In addition, while traditional LLN applications like environment monitoring can be supported by unreliable UDP, certain applications do require high throughput and reliable delivery. Some examples are high-throughput sensing applications like anemometry (Section 5.3.2) and vibration monitoring [259]. Even low-throughput sensing applications could benefit from high throughput and reliability for firmware updates and for configuration and management (e.g., over Telnet or SSH). Applications built on expressive cryptography (e.g., JEDI in Chapter 7) would also benefit from high throughput and reliability to transfer large keys and ciphertexts over the network. TCP, *if it performs well in LLNs*, could benefit these applications.

Adopting TCP in LLNs may also open an interesting research agenda for IoT. TCP is the default transport protocol outside of LLNs, and history has shown that, to justify other transport protocols, application characteristics must offer substantial opportunity for optimization (e.g., [485,



(a) Anemometer.

(b) Hamilton-based PCB (bottom and top).

Figure 5.1: Hamilton-based ultrasonic anemometer.

486, 177]). If TCP becomes a viable option in LLNs, it would raise the bar for application-specific LLN protocols, resulting in some potentially interesting alternatives.

Although adopting TCP in LLNs could yield significant benefit and an interesting agenda, its feasibility and performance remain in question. This motivates our study.

5.3.2 Anemometry: An Example TCP-Based LLN Application

To demonstrate the benefits of TCP in LLNs, we describe *anemometry in buildings*, an LLN application that benefits from high throughput and reliable delivery.

An *anemometer* is a sensor that measures air velocity. Anemometers may be deployed in a building to diagnose problems with the Heating, Ventilation, and Cooling system (HVAC), and also to collect air flow measurements for improved HVAC control. This requires anemometers in difficult-to-reach locations, such as in air flow ducts, where it is infeasible to run wires. Therefore, anemometers must be battery-powered and must transmit readings wirelessly, making LLNs attractive.

We used anemometers based on the Hamilton platform [18], each consisting of four ultrasonic transceivers arranged as vertices of a tetrahedron (Figure 5.1). To measure the air velocity, each transceiver, in turn, emits a burst of ultrasound, and the impulse is measured by the other three transceivers. This process results in a total of 12 measurements.

Calculating the air velocity from these measurements is computationally infeasible on the anemometer itself, because Hamilton does not have hardware floating point support and the computations require complex trigonometry. Measurements must be transmitted over the network to a server that processes the data. Furthermore, a specific property of the analytics is that it requires a contiguous stream of data to maintain calibration (a numerical integration is performed on the measurements). Thus, the application requires a high sample rate (1 Hz), and is sensitive to data loss. A protocol for *reliable* delivery, like TCP or CoAP, is therefore necessary.

We note that the 1 Hz sample rate for this application is much higher than the sample rate of most sensors deployed in buildings. For example, a sensor measuring temperature, humidity, or occupancy in a building typically only generates a single reading every few tens of seconds or every few minutes. Furthermore, each individual reading from the anemometer is quite large (82 bytes), given that it encodes all 12 measurements (plus a small header). Given the higher data rate requirements of the anemometer application, it is natural to use a higher-capacity battery than the standard AA batteries used in most motes. The higher cost of such a battery is justified by the higher cost of the anemometer transducers.

5.4 Empirical Methodology

This section presents our methodology, carefully chosen to ground our study of full-scale TCP in LLNs.

5.4.1 Network Stack

Transport layer. That only a few full-scale TCP stacks exist, with a body of literature covering decades of refining, demonstrates that developing a feature-complete implementation of TCP is complex and error-prone [371]. Using a well-tested TCP implementation would ensure that results from our measurement study are due to the TCP *protocol*, not an artifact of the TCP *implementation* we used. Thus, we leverage the TCP implementation in FreeBSD 10.3 [385] to ground our study. We ported it to run in embedded operating systems and resource-constrained embedded devices (Section 5.4.2).

To verify the effectiveness of full-scale TCP in LLNs, we compare with CoAP [426], CoCoA [52], and unreliable UDP. CoAP is a standard LLN protocol that provides reliability on top of UDP. It is the most promising LLN alternative to TCP, gaining momentum in both academia [123, 460, 285, 408, 52, 413] and industry [250, 258], with adoption by Cisco [149, 433], Nest/-Google [360], and Arm [425]. CoCoA [52] is a recent proposal that augments CoAP with RTT estimation.

It is attractive to compare TCP to a variety of commercial systems, as has been done by a number of studies in LTE/WLANs [486, 177]. Unfortunately, multihop LLNs have not yet reached the level of maturity to support a variety of commercial offerings; only CoAP has an appreciable level of commercial adoption. Other protocols are research proposals that often (1) are implemented for now-outdated operating systems and hardware or exist only in simulation [247, 279, 5], (2) target a very specific application paradigm [465, 438, 492], and/or (3) do not use IP [465, 247, 279, 366]. We choose CoAP and CoCoA because they are not subject to these constraints.

Layers 1 to 3. Because it is burdensome to place a border router with LAN connectivity within wireless range of every low-power host (e.g., sensor node), it is common to transmit data (e.g., readings) over *multiple* wireless LLN hops. Although each sensor must be battery-powered, it

	TelosB	Hamilton	Firestorm	Raspberry Pi
CPU	MSP430	Cortex-M0+	Cortex-M4	Cortex-A53
RAM	10 KiB	32 KiB	64 KiB	256 MB
ROM	48 KiB	256 KiB	512 KiB	SD Card

Table 5.2: Comparison of the platforms we used (Hamilton and Firestorm) to TelosB and Raspberry Pi.

is reasonable to have a wall-powered LLN router node within transmission range of it.[1] This motivates Thread[2] [450, 278], a recently developed protocol standard that constructs a multihop LLN over IEEE 802.15.4 links with *wall-powered, always-on* router nodes and *battery-powered, duty-cycled* leaf nodes. We use OpenThread [360], an open-source implementation of Thread.

Thread decouples routing from energy efficiency. Among routers, it provides a mesh topology, frequent route updates, and asymmetric bidirectional routing for reliability. Each *leaf node* duty cycles its radio and simply chooses a router with good link quality, called its *parent*, as its next hop to all other nodes. The duty cycling uses *listen-after-send* [409]. A leaf node's parent stores downstream packets destined for that leaf node, until the leaf node sends it a *data request* message. A leaf node, therefore, can keep its radio powered off most of the time; infrequently, it sends a data request message to its parent, and turns on its radio for a short interval afterward to listen for downstream packets queued at its parent. Leaf nodes may send upstream traffic at any time. Each node uses CSMA-CA for medium access.

5.4.2 Embedded Hardware

We use two embedded hardware platforms: Hamilton [274] and Firestorm [14]. Hamilton uses a SAMR21 SoC with a 48 MHz Cortex-M0+, 256 KiB of ROM, and 32 KiB of RAM. Firestorm uses a SAM4L 48 MHz Cortex-M4 with 512 KiB of ROM and 64 KiB of RAM. While these platforms are more powerful than the TelosB [380], an older LLN platform widely used in past studies, they are heavily resource-constrained compared to a Raspberry Pi (Table 5.2). Both platforms use the AT86RF233 radio, which supports IEEE 802.15.4. We use its standard data rate, 250 kb/s. We use Hamilton/OpenThread in our experiments; for comparison, we provide some results from Firestorm and other network stacks in Section 5.5 and Section 5.6.3.

Handling automatic radio features. The AT86RF233 radio has built-in hardware support for link-layer retransmissions and CSMA-CA. However, it automatically enters low-power mode during CSMA backoff, during which it does not listen for incoming frames [324]. This behavior, which we call *deaf listening*, interacts poorly with TCP when radios are always on, because TCP

[1]The assumption of powered "core routers" is reasonable for most IoT use cases, which are typically indoors. Recent IoT protocols, such as Thread [450] and BLEmesh [59], take advantage of powered core routers.

[2]Thread has a large amount of industry support with a consortium already consisting of over 100 members [449], and is used in real IoT products sold by Nest/Google [448]. Given this trend, using Thread makes our work timely.

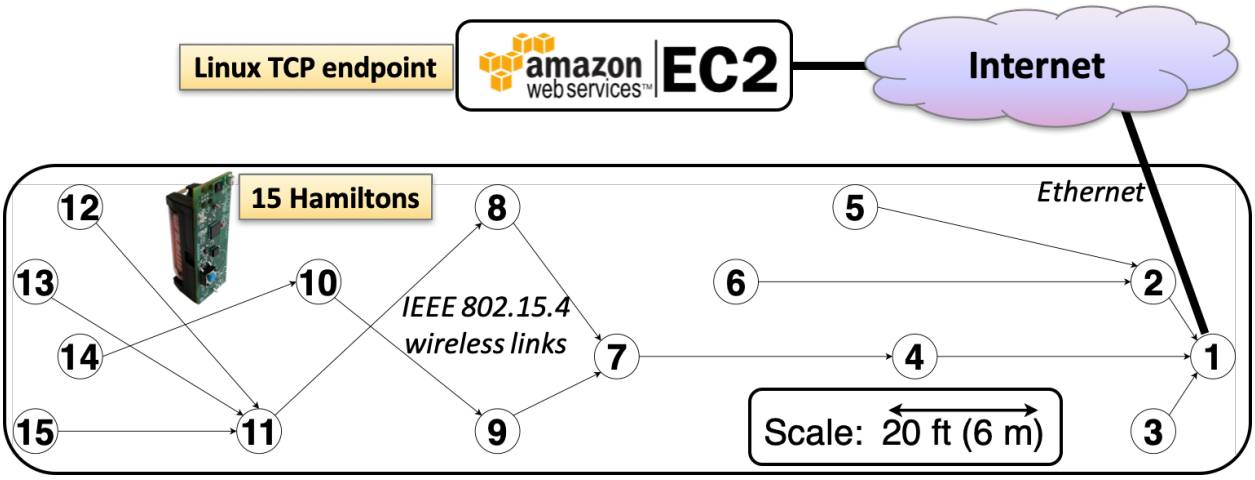


Figure 5.2: Snapshot of uplink routes in OpenThread topology at transmission power of -8 dBm (5 hops). Node 1 is the border router with Internet connectivity.

requires bidirectional flow of packets—data in one direction and ACKs in the other. This may initially seem concerning, as deaf listening is an important power-saving feature. Fortunately, this problem disappears when using OpenThread's listen-after-send duty-cycling protocol, as leaf nodes never transmit data when listening for downstream packets. For experiments with always-on radios, we do not use the radio's capability for hardware CSMA and link retries; instead, we perform these operations in software.

Multihop Testbed. We construct an indoor LLN testbed, depicted in Figure 5.2, with 15 Hamiltons where node 1 is configured as the border router. OpenThread forms a 3-to-5-hop topology at transmission power of -8 dBm. Embedded TCP endpoints (Hamiltons) communicate with a Linux TCP endpoint (server on Amazon EC2) via the border router. During working hours, interference is present in the channel, due to people in the space using Wi-Fi and Bluetooth devices in the 2.4 GHz frequency band. At night, when there are few/no people in the space, there is much less interference.

5.5 Implementation of *TCPlp*

We seek to answer the following two questions: (1) Does full-scale TCP fit within the limited memory of modern LLN platforms? (2) How can we integrate a TCP implementation from a traditional OS into an embedded OS? To this end, we develop a TCP stack for LLNs based on the TCP implementation in FreeBSD 10.3, called *TCPlp* [292], on multiple embedded operating systems, RIOT OS [29] and TinyOS [309]. We use *TCPlp* in our measurement study in future sections.

Although we carefully preserved the protocol logic in the FreeBSD TCP implementation, achieving correct and performant operation on sensor platforms was a nontrivial effort. We had to

	uIP	BLIP	GNRC	*TCPlp*
Flow Control	Yes	Yes	Yes	Yes
Congestion Control	N/A	No	Yes	Yes
RTT Estimation	Yes	No	Yes	Yes
MSS Option	Yes	No	Yes	Yes
OOO Reassembly	No	No	Yes	Yes
TCP Timestamps	No	No	No	Yes
Selective ACKs	No	No	No	Yes
Delayed ACKs	No	No	No	Yes

Table 5.3: Comparison of features among embedded TCP stacks: uIP (Contiki), BLIP (TinyOS), GNRC (RIOT), and *TCPlp (our work)*.

modify the FreeBSD TCP implementation according to the concurrency model of each embedded network stack and the timer abstractions provided by each embedded operating system. We also made other modifications to the FreeBSD TCP implementation to reduce its memory footprint.

We first discuss the TCP features supported by our implementation. Then we discuss and describe our changes to the FreeBSD TCP implementation below, with attention to the resulting memory overhead of TCP.

5.5.1 Supported TCP Features

TCPlp includes features from FreeBSD that improve standard communication, like a sliding window, New Reno congestion control, zero-window probes, delayed ACKs, selective ACKs, TCP timestamps, and header prediction. Table 5.3 compares the feature set of *TCPlp* to features in embedded TCP stacks. The TCP implementations in uIP and BLIP lack features core to TCP. uIP allows only one unACKed in-flight segment, eschewing TCP's sliding window. BLIP does not implement RTT estimation or congestion control. The TCP implementation in GNRC lacks features such as TCP timestamps, selective ACKs, and delayed ACKs, which are present in most full-scale TCP implementations. Another comparison of features in TCP stacks is available in the slides from an OpenThread developer meeting [294].

In addition to supporting the protocol-level features summarized in 5.3, *TCPlp* is likely more robust than other embedded TCP stacks because it is based on a well-tested TCP implementation. While seemingly minor, some details, implemented incorrectly by TCP stacks, have had important consequences for TCP's behavior [371]. *TCPlp* benefits from a thorough implementation of each aspect of TCP.

For example, *TCPlp*, by virtue of using the FreeBSD TCP implementation, benefits from a robust implementation of congestion control. *TCPlp* implements not only the basic New Reno algorithm, but also Explicit Congestion Notification [175], Appropriate Byte Counting [7, 8] and Limited Transmissions [9]. It also inherits from FreeBSD heuristics to identify and correct "bad

retransmissions" (as in Section 2.8 of [11]): if, after a retransmission, the corresponding ACK is received very soon (within $\frac{\text{RTT}}{2}$ of the retransmission), the ACK is assumed to correspond to the originally transmitted segment as opposed to the retransmission. The FreeBSD implementation and *TCPlp* recover from such "bad retransmissions" by restoring `cwnd` and `ssthresh` to their former values before the packet loss. Aside from congestion control, *TCPlp* benefits from header prediction [120], which introduces a "fast code path" to process common-case TCP segments (in-sequence data and ACKs) more efficiently, and Challenge ACKs [390], which make it more difficult for an attacker to inject an RST into a TCP connection.

Enhancements such as these make us more confident that our observed results are fundamental to TCP, as opposed to artifacts of poor implementation. Furthermore, they allow us to focus on performance problems arising from the challenges of LLNs, as opposed to general TCP-related challenges that the research community has already solved in the context of traditional networks and operating systems.

TCPlp, however, omits some features in FreeBSD's TCP/IP stack. We omit dynamic window scaling, as buffers large enough to necessitate it ($\geq$ 64 KiB) would not fit in memory. We omit the urgent pointer, as it not recommended for use [198] and would only complicate buffering. Certain security features, such as host cache, TCP signatures, SYN cache, and SYN cookies are outside the scope of this work. As mentioned above, however, we do retain Challenge ACKs [390].

5.5.2 Concurrency Model

We describe how the concurrency model of the underlying system interacts with *TCPlp*.

5.5.2.1 GNRC and OpenThread (RIOT OS)

RIOT OS provides threads as the basic unit of concurrency. Asynchronous interaction with hardware is done by interrupt handlers that preempt the current thread, perform a short operation in the interrupt context, and signal a related thread to perform any remaining operation outside of interrupt context. Then the thread is placed on the RIOT OS scheduler queue and is scheduled for execution depending on its priority.

The GNRC network stack for RIOT OS runs each network layer (or module) in a separate thread. Each thread has a priority and can be preempted by a thread with higher priority or by an interrupt. The thread for a lower network layer has higher priority than the thread for a higher layer.

The port of OpenThread for RIOT OS on which we implemented *TCPlp* handles received packets in one thread and sends packets from another thread, where the thread for received packets has higher priority [274]. The rationale for this design is to ensure timely processing of received packets at the radio, which is especially important in the context of a high-throughput flow.

To adapt *TCPlp* for GNRC, we run the FreeBSD implementation as a single TCP-layer thread, whose priority is between that of the application-layer thread and the IPv6-layer thread. To adapt *TCPlp* for OpenThread on RIOT OS, we call the TCP protocol logic (`tcp_input()`) at the appropriate point along the receive path, and send packets from the TCP protocol logic (`tcp_output()`)

using the established send path. As explained in Section 5.5.3, we also use an additional thread for timer callbacks in RIOT OS.

Given that TCP state can be accessed concurrently from multiple threads—the TCP thread (GNRC) or receive thread (OpenThread), the application threads, and timer callbacks—we needed to synchronize access to it. The FreeBSD implementation allows fine-grained locking of connection state to allow different connections to be serviced in parallel on different CPUs. Given that low-power embedded sensors typically have only one CPU, however, we opted for simplicity, instead using a single global TCP lock for *TCPlp*.

5.5.2.2 BLIP (TinyOS) and Standalone OpenThread

TinyOS uses an event-driven concurrency model based on split-phase operations, consisting of an event loop that executes on a *single* stack. For concurrency, TinyOS provides three types of unique operations: *commands* and *events*, which are executed immediately, and *tasks*, which are scheduled for execution after all preceding tasks are completed. An interrupt handler may preempt the current function, perform a short operation in the interrupt context using *asynchronous* events and commands, and *post* a task to perform any remaining computation later. To adapt the thread-based FreeBSD implementation to the event-driven TinyOS, we execute the primary functions of FreeBSD, such as `tcp_output()` and `tcp_input()`, within *tasks* outside of interrupt context. Because tasks in TinyOS cannot preempt each other, we remove the locking present in the FreeBSD TCP implementation.

Later, we integrated *TCPlp* directly into OpenThread. This implementation is capable of running in a standalone OpenThread-based system without RIOT OS (Section 9.1.1). OpenThread uses event-driven concurrency similar to TinyOS, exposed to the network stack as `Tasklets`, so our concurrency design for *TCPlp* in OpenThread is similar to our concurrency design for *TCPlp* in TinyOS.

5.5.3 Timer Event Management

Given that many TCP operations are based on timer events, achieving correct timer operation is important. For example, if an RTO timer event is dropped by the embedded operating system, the RTO timer will not be rescheduled, and the connection may hang.

For a simple and stable operation, many existing embedded TCP stacks, including the uIP, lwIP, and BLIP TCP stacks, rely on a periodic, fixed-interval clock in order to check for expired timeouts. Instead, *TCPlp* uses one-shot tickless timers as FreeBSD 10.3 does [246], which is beneficial in two ways: (1) When there are no scheduled timers, the tickless timers allow the CPU to sleep, rather than being needlessly woken up at a fixed interval, resulting in lower energy consumption [274]. (2) Unlike fixed periodic timers, which can only be serviced on the next tick after they expire, tickless timers can be serviced as soon as they expire. To obtain these advantages, however, an embedded operating system must robustly manage asynchronous timer callbacks.

TinyOS has a single event queue maintained by the scheduler. The semantics of TinyOS guarantee that a task can exist in the event queue only once, even if it is *posted* (i.e., scheduled for

	Protocol	Event Scheduler	User Library
ROM	21352 B	1696 B	5384 B
RAM (Per Active Socket)	488 B	40 B	36 B
RAM (Per Passive Socket)	16 B	16 B	36 B

Table 5.4: Memory usage of *TCPlp* on TinyOS. Our *TCPlp* implementation spans three modules: (1) protocol implementation, (2) event scheduler that injects callbacks into userspace, and (3) user library.

execution) multiple times before executing. Therefore, the event queue can be sized appropriately at compile-time to not overflow. Furthermore, TinyOS handles received packets in a separate queue than tasks. This ensures that TCP timer callbacks will not be dropped.

This is not the case for RIOT OS. Timer callbacks either handle the timer entirely in interrupt context, or put an event on a thread's message queue, so that the thread performs the required callback operation. Each network protocol supported by RIOT OS has a single thread. Because a thread's message queue in RIOT OS is used to hold both received packets and timer events, there is no guarantee when a timer expires that there is enough space in the thread message queue to accept a timer event; if there is not enough space, RIOT OS drops the timer event. Furthermore, if a timer expires multiple times before its event is handled by the thread, multiple events for the same timer can exist simultaneously in the queue; *we cannot find an upper bound on the number of slots in the message queue used by a single timer*. To provide robust TCP operation on RIOT OS, we create a second thread used exclusively for TCP timers. We handle timers similarly to TinyOS' *post* operation, by preventing the message queue from having multiple callback events of a single timer. This eliminates the possibility of timer event drops.

5.5.4 Connection State for *TCPlp*

We use separate structures for *active sockets* used to send and receive bytes, and *passive sockets* used to listen for incoming connections, as passive sockets require less memory.

Table 5.4 and Table 5.5 depict the memory footprint of *TCPlp* on TinyOS and RIOT OS. The memory required for the protocol and application state of an active TCP socket fits in a few hundred bytes, less than 1% of the available RAM on the Cortex-M4 (Firestorm) and 2% of that on the Cortex-M0+ (Hamilton). Although *TCPlp* includes heavyweight features not traditionally included in embedded TCP stacks, it fits well within available memory.

5.5.5 Memory-Efficient Data Buffering

Existing embedded TCP stacks, such as uIP and BLIP, allow *only one TCP packet in the air*, eschewing careful implementation of send and receive buffers [277]. These buffers, however, are key to supporting TCP's sliding window functionality. We observe in Section 5.6.2 that *TCPlp* performs well with only 2-3 KiB send and receive buffers, which comfortably fit in memory even

	Protocol	Socket Layer	`posix_sockets`
ROM	19972 B	6216 B	5468 B
RAM (Per Active Socket)	364 B	88 B	48 B
RAM (Per Passive Socket)	12 B	88 B	48 B

Table 5.5: Memory usage of *TCPlp* on RIOT OS. We also include RIOT's `posix_sockets` module, used by *TCPlp* to provide a Unix-like interface.

when naïvely preallocated at compile time. Given that buffers dominate *TCPlp*'s memory usage, however, we discuss techniques to optimize their memory usage.

5.5.5.1 Send Buffer: Zero-Copy

Zero-copy techniques [49, 146, 447, 316, 326] are typically used in situations where the time for the CPU to copy memory is a significant bottleneck. Our situation is very different; the radio, not the CPU, is the bottleneck, owing to the low bandwidth of IEEE 802.15.4. By using a zero-copy send buffer, however, we can avoid allocating memory to intermediate buffers that would otherwise be needed to copy data, thereby reducing the network stack's total memory usage.

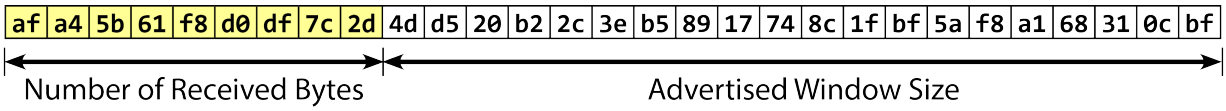
In TinyOS, for example, the BLIP network stack supports vectored I/O; an outgoing packet passed to the IPv6 layer is specified as an `iovec`. Instead of allocating memory in the packet heap for each outgoing packet, *TCPlp* simply creates `iovecs` that point to existing data in the send buffer. This decreases the required size of the packet heap.

Unfortunately, zero-copy optimizations were not possible for the OpenThread implementation, because OpenThread does not support vectored I/O for sending packets. The result is that the *TCPlp* implementation requires a few kilobytes of additional memory for the send buffer on this platform.

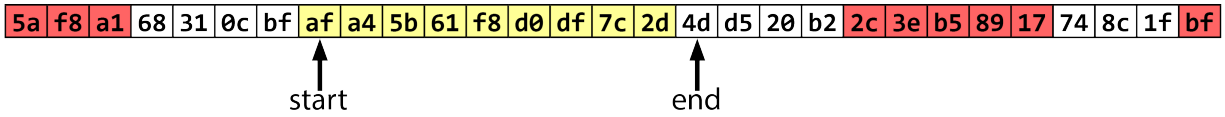
5.5.5.2 Receive Buffer: In-Place Reassembly Queue

Not all zero-copy optimizations are useful in the embedded setting. In FreeBSD, received packets are passed to the TCP implementation as `mbufs` [489]. The receive buffer and reassembly buffer are `mbuf` chains, so data need not be copied out of `mbufs` to add them to either buffer or recover from out-of-order delivery. Furthermore, buffer sizes are chosen dynamically [414], and are merely a *limit* on their actual size. In our memory-constrained setting, such a design is dangerous because its memory usage is nondeterministic; there is additional memory overhead, due to headers, if the data are delivered in many small packets instead of a few large ones.

We opted for a flat array-based circular buffer for the receive buffer in *TCPlp*, primarily owing to its determinism in a limited-memory environment: buffer space is reserved at *compile time*. Head/tail pointers delimit which part of the array stores in-sequence data. To reduce memory consumption, we store out-of-order data in the same receive buffer, at the same position as if they were in-sequence. We use a bitmap, not head/tail pointers, to record where out-of-order data are



(a) Naïve receive buffer. Note that size of advertised window + size of buffered data = size of receive buffer.



(b) Receive buffer with in-place reassembly queue. In-sequence data (yellow) is kept in a circular buffer, and out-of-order segments (red) are written in the space past the received data.

Figure 5.3: Naïve and final TCP receive buffers.

	Fast Ethernet	Wi-Fi	Ethernet	IEEE 802.15.4
Capacity	100 Mb/s	54 Mb/s	10 Mb/s	250 kb/s
MTU	1500 B	1500 B	1500 B	104–116 B
Transmission Time	0.12 ms	0.22 ms	1.2 ms	4.1 ms

Table 5.6: Comparison of TCP/IP links.

Header	IEEE 802.15.4	6LoWPAN	IPv6	TCP	Total
1st Frame	11–23 B	5 B	2–28 B	20–44 B	38–107 B
*n*th Frame	11–23 B	5–12 B	0 B	0 B	16–35 B

Table 5.7: Header overhead with 6LoWPAN fragmentation.

stored, because out-of-order data need not be contiguous. We call this an *in-place reassembly queue* (Figure 5.3).

5.6 TCP in a Low-Power Network

In this section, we characterize how full-scale TCP interacts with a low-power network stack, resource-constrained hardware, and a low-bandwidth link.

5.6.1 Reducing Header Overhead using MSS

In traditional networks, it is customary to set the Maximum Segment Size (MSS) to the link MTU (or path MTU) minus the size of the TCP/IP headers. IEEE 802.15.4 frames, however, are *an order of magnitude smaller* than frames in traditional networks (Table 5.6). The TCP/IP headers

consume more than half of the frame's available MTU. As a result, TCP performs poorly, incurring more than 50% header overhead.

Earlier approaches to running TCP over low-MTU links (e.g., low-speed serial links) have used TCP/IP header compression based on per-flow state [248] to reduce header overhead. In contrast, the 6LoWPAN adaptation layer [347], designed for LLNs, supports only *flow-independent* compression of the IPv6 header using shared link-layer state, a clear departure from per-flow techniques. A key reason for this is that the compressor and decompressor in an LLN (host and border router) are separated by several IP hops,[3] making it desirable for intermediate nodes to be able to determine a packet's IP header without per-flow context (see Section 10 of [347]).

That said, compressing TCP headers separately from IP addresses using per-flow state is a promising approach to further amortize header overhead. There is preliminary work in this direction [26, 27], but it is based on uIP, which has one in-flight segment, and does not fully specify how to resynchronize compression state after packet loss with a multi-segment window. It is also not officially standardized by the IETF.

Therefore, we take an approach orthogonal to header compression. We instead choose an MSS larger than the link MTU admits, *relying on fragmentation at the lower layers to decrease header overhead.* Fragmentation is handled by 6LoWPAN, which acts at Layer 2.5, between the link and network layers. Unlike end-to-end IP fragmentation, the 6LoWPAN fragments exist only within the LLN, and are reassembled into IPv6 packets when leaving the network.

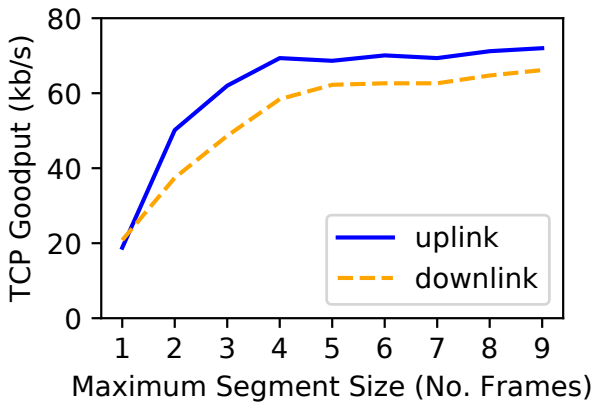
Relying on fragmentation is effective because, as shown in Table 5.7, TCP/IP headers consume space in the first fragment, but not in subsequent fragments. Using an excessively large MSS, however, decreases reliability because the loss of one fragment results in the loss of an entire packet. Existing work [25] has identified this trade-off and investigated it in simulation in the context of power consumption. We investigate it in the context of goodput in a live network.

Figure 5.4a shows the bandwidth as the MSS varies. As expected, we see poor performance at a small MSS due to header overhead. Performance gains diminish when the MSS becomes larger than 5 frames. We recommend using an MSS of about 5 frames, but it is reasonable to decrease it to 3 frames if more wireless loss is expected. **Despite the small frame size of IEEE 802.15.4, we can effectively amortize header overhead for TCP using an atypical MSS.** Adjusting the MSS is orthogonal to TCP header compression. We hope that widespread use of TCP over 6LoWPAN, perhaps based on our work, will cause TCP header compression to be separately investigated and possibly used together with a large MSS.

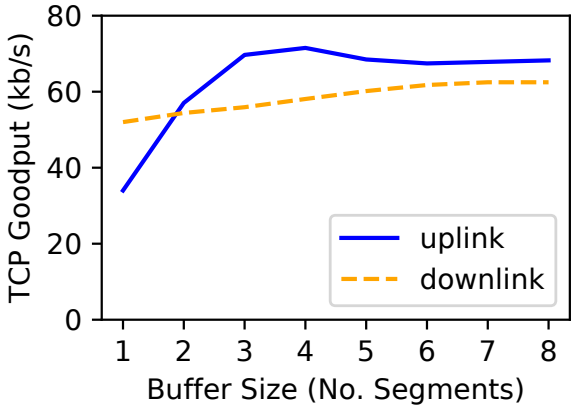
5.6.2 Impact of Buffer Size

Whereas simple TCP stacks, like uIP, allow only one in-flight segment, full-scale TCP requires complex buffering (Section 5.5.5). In this section, we vary the size of the buffers (send buffer for uplink experiments and receive buffer for downlink experiments) to study how it affects the bandwidth. In varying the buffer size, we are directly affecting the size of TCP's flow window. We

[3]Thread deliberately does not abstract the mesh as a single IP link. Instead, it organizes the LLN mesh as a set of *overlapping link-local scopes*, using IP-layer routing to determine the path packets take through the mesh [233].



(a) Effect of varying MSS.



(b) Effect of varying buffer size.

Figure 5.4: TCP goodput over one IEEE 802.15.4 hop.

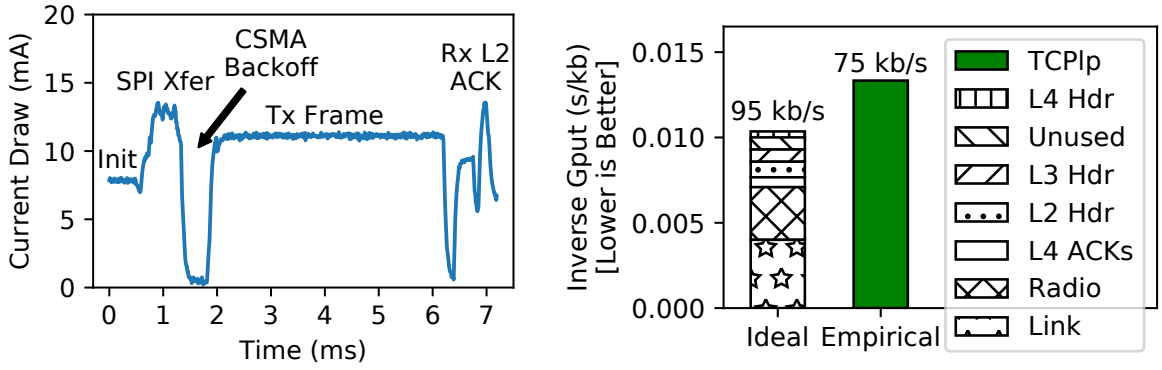
expect throughput to increase with the flow window size, with diminishing returns once it exceeds the bandwidth-delay product (BDP). The result is shown in Figure 5.4b. **Goodput levels off at a buffer size of 3 to 4 segments (1386 B to 1848 B), indicating that the buffer size needed to fill the BDP fits comfortably in memory.** Indeed, the BDP in this case is about $125\text{kb/s} \cdot 0.1\text{s} \approx 1.6\text{KiB}$.[4]

Downlink goodput at a buffer size of one segment is unusually high. This is because FreeBSD does not delay ACKs if the receive buffer is full, reducing the effective RTT from ≈ 130 ms to ≈ 70 ms. Indeed, goodput is very sensitive to RTT when the buffer size is small, because TCP exhibits "stop-and-wait" behavior due to the small flow window.

5.6.3 Direct TCP Connection

We also consider TCP goodput between two embedded nodes over the IEEE 802.15.4 link, over a single hop without any border router. Using the OpenThread network stack with RIOT OS on Hamilton, we are able to achieve 75 kb/s over a single TCP connection. For comparison, we are able to achieve 63 kb/s goodput over a TCP connection between two Hamilton motes using RIOT's GNRC network stack, and 71 kb/s using the BLIP stack on Firestorm. Later, we integrated *TCPlp* directly into OpenThread (Section 9.1.1); with the resulting implementation, we are able to achieve approximately 80 kb/s between two nRF52840-DK boards. **This suggests that our results are reproducible across multiple platforms and embedded network stacks.** The minor performance degradation in GNRC is possibly explained by its greater header overhead due to implementation differences, and by its IPC-based thread-per-layer concurrency architecture, which

[4]We estimate the bandwidth as 125 kb/s rather than 250 kb/s to account for the radio overhead identified in Section 5.6.4.



(a) Unicast of a single frame, measured with an oscilloscope.

(b) *TCPlp* goodput compared with raw link bandwidth and overheads.

Figure 5.5: Analysis of overhead limiting *TCPlp*'s goodput.

has known inefficiencies [119]. This suggests that the implementation of the underlying network stack, particularly with regard to concurrency, could affect TCP performance in LLNs.

5.6.4 Upper Bound on Single-Hop Goodput

We consider consider TCP goodput over a single hop without any border router, as we did in Section 5.6.3. Figure 5.5b lists various sources of overhead that limit *TCPlp*'s goodput, along with the ideal upper bounds that they admit. **Link** overhead refers to the 250 kb/s link capacity. **Radio** overhead includes SPI transfer to/from the radio (i.e., packet copying [362]), CSMA, and link-layer ACKs, which cannot be pipelined because the AT86RF233 radio has only one frame buffer. A full-sized 127-byte frame spends 4.1 ms in the air at 250 kb/s, but the radio takes 7.2 ms to send it (Figure 5.5a), almost halving the link bandwidth available to a single node. This is consistent with prior results [362]. **Unused** refers to unused space in link frames due to inefficiencies in the 6LoWPAN implementation. Overall, we estimate a 95 kb/s upper bound on goodput (100 kb/s without TCP headers). Our 75 kb/s measurement is within 25% of this upper bound, substantially higher than prior work (Table 5.8). The difference from the upper bound is likely due to network stack processing and other real-world inefficiencies.

[5]For this study, we list aggregate goodput over multiple TCP flows.

[6]One study [155] achieves ≈ 16 kb/s over multiple hops using the Linux TCP stack. We do not include it in Table 5.8 because it does not capture the resource constraints of LLNs (it uses traditional computers for the end hosts) and does not consider hidden terminals (it uses different wireless channels for different wireless hops). It uses TCP to evaluate link-layer burst forwarding.

	[513]	[26]	[219]	[277][5]	[233, 232]	Our Work (Hamilton Platform)
TCP Stack	uIP	uIP	uIP	BLIP	Arch Rock	*TCPlp* (RIOT OS, OpenThread)
Maximum Segment Size	1 Frame	1 Frame	4 Frames	1 Frame	1024 bytes	5 Frames
Window Size	1 Segment	1 Segment	1 Segment	1 Segment	1 Segment	1848 bytes (4 Segments)
Goodput (One Hop)	1.5 kb/s	$\approx$ 13 kb/s	$\approx$ 12 kb/s	$\approx$ 4.8 kb/s	15 kb/s	75 kb/s
Goodput (Multi-Hop)	$\approx$ 0.55 kb/s	$\approx$ 6.4 kb/s	$\approx$ 12 kb/s	$\approx$ 2.4 kb/s	9.6 kb/s	20 kb/s

Table 5.8: Comparison of *TCPlp* to existing TCP implementations used in network studies over IEEE 802.15.4 networks.[6] Goodput figures obtained by reading graphs in the original paper (rather than stated numbers) are marked with the $\approx$ symbol.

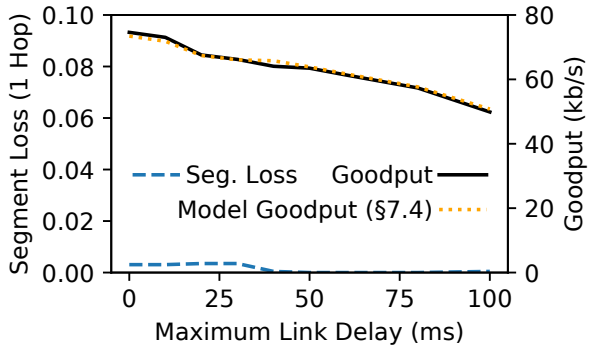
5.7 TCP Over Multiple Wireless Hops

We instrument TCP connections between Hamilton nodes in our multi-hop testbed, without using the EC2 server.

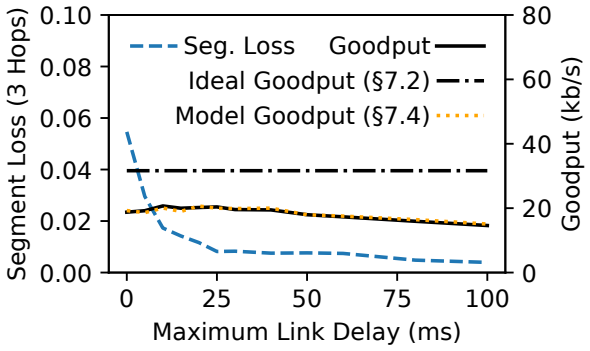
5.7.1 Mitigating Hidden Terminals in LLNs

Prior work over traditional WLANs has shown that hidden terminals degrade TCP performance over multiple wireless hops [187]. Using RTS/CTS for hidden terminal avoidance has been shown to be effective in WLANs. This technique has an unacceptably high overhead in LLNs [488], however, because data frames are small (Table 5.6), comparable in size to the additional control frames required. Prior work in LLNs has carefully designed application traffic, using rate control [279, 234] and link-layer delays [488], to avoid hidden terminals.

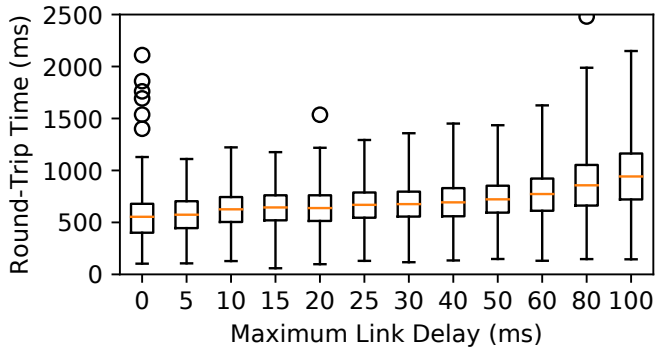
But prior work does not explore these techniques in the context of TCP. Unlike protocols like CoAP and simplified TCP implementations like uIP, a full-scale TCP flow has a *multi-segment sliding window* of unacknowledged data, making it unclear *a priori* whether existing LLN techniques will be sufficient. In particular, rate control seems sufficient because of bi-directional packet flow in TCP (data in one direction and ACKs in the other). So, rather than applying rate control, we attempt to avoid hidden terminals by adding a delay d between link-layer retries in addition to CSMA backoff. After a failed link transmission, a node waits for a random duration between 0 and d, before retransmitting the frame. The idea is that if two frames collide due to a hidden terminal, the delay will prevent their link-layer retransmissions from colliding.



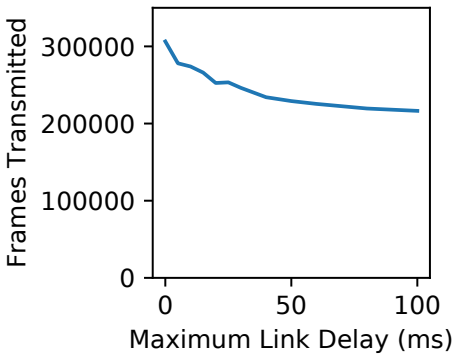
(a) TCP goodput, one hop.



(b) TCP goodput, three hops.



(c) RTT, three hops (outliers omitted).



(d) Total frames sent, three hops.

Figure 5.6: Effect of varying time between link-layer retransmissions. Reported "segment loss" is the loss rate of TCP segments, not individual IEEE 802.15.4 frames. It includes only losses not masked by link-layer retries.

We modified OpenThread, which previously had no delay between link retries, to implement this. As expected, single-hop performance (Figure 5.6a) decreases as the delay between link retries increases; hidden terminals are not an issue in that setting. Packet loss is high for the multihop experiment (Figure 5.6b) when the link retry delay is 0, as is expected from hidden terminals. **Adding a small delay between link retries, however, effectively reduces packet loss.** Making the delay too large raises the RTT (Figure 5.6c).

We prefer a smaller frame/segment loss rate, even if goodput stays the same, in order to make more efficient use of network resources. Therefore, we prefer a moderate delay ($d = 40$ ms) to a small delay ($d = 5$ ms), even though both provide the same goodput, because the frame and segment loss rates are smaller when d is large (Figures 5.6b and 5.6d).

5.7.2 Upper Bound on Multi-Hop Goodput

Comparing Figures 5.6a and 5.6b, goodput over three wireless hops is substantially smaller than goodput over a single hop. Prior work has observed similar throughput reductions over multiple hops [277, 362]. It is due to radio scheduling constraints inherent in the multihop setting, which we describe in this section. Let B be the bandwidth over a single hop.

Consider a two-hop setup: $S \to R_1 \to D$. R_1 cannot receive a frame from S while sending a frame to D, because its radio cannot transmit and receive simultaneously. Thus, the maximum achievable bandwidth over two hops is $\frac{B}{2}$.

Now consider a three-hop setup: $S \to R_1 \to R_2 \to D$. By the same argument, if a frame is being transferred over $R_1 \to R_2$, then neither $S \to R_1$ nor $R_2 \to D$ can be active. Furthermore, if a frame is being transferred over $R_2 \to D$, then R_1 can hear that frame. Therefore, $S \to R_1$ cannot transfer a frame at that time; if it does, then its frame will collide at R_1 with the frame being transferred over $R_2 \to D$. Thus, the maximum bandwidth is $\frac{B}{3}$. We depict this ideal upper bound in Figure 5.6b, taking B to be the ideal single-hop goodput from Section 5.6.4.

In setups with more than three hops, every set of three adjacent hops is subject to this constraint. The first hop and fourth hop, however, may be able to transfer frames simultaneously. Therefore, the maximum bandwidth is still $\frac{B}{3}$. In practice, goodput may fall slightly because transmissions from a node may *interfere* with nodes multiple hops away, even if they can only be received by its immediate neighbors.

We made empirical measurements with $d = 40$ ms to validate this analysis. Goodput over one hop was 64.1 kb/s; over two hops, 28.3 kb/s; over three hops, 19.5 kb/s; and over four hops, 17.5 kb/s. This roughly fits the model.

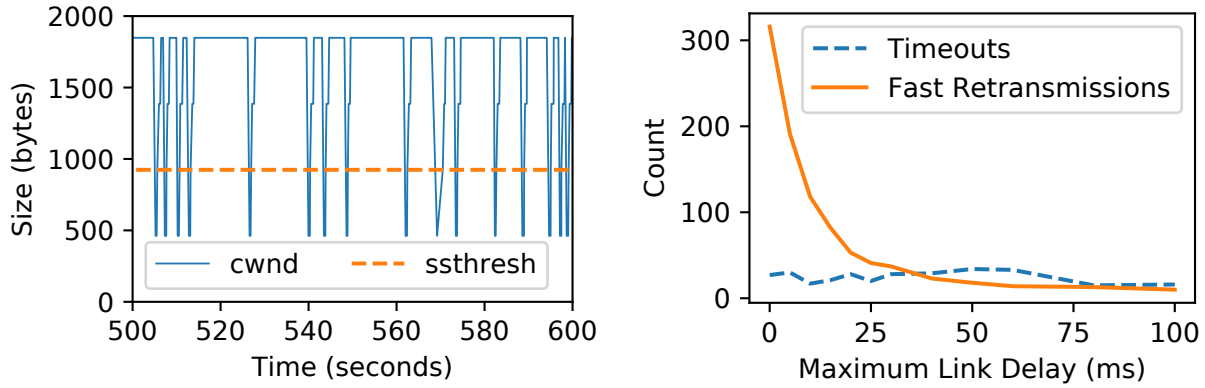
This analysis justifies why the same window size works well for both the one-hop experiments and the three-hop experiments in Section 5.7.1. Although the RTT is three times higher, the bandwidth-delay product is approximately the same. **Crucially, this means that the 2 KiB buffer size we determined in Section 5.6.2, which fits comfortably in memory, remains applicable for up to three wireless hops.**

5.7.3 TCP Congestion Control in LLNs

Recall that small send/receive buffers of only 1848 bytes (4 TCP segments) each are enough to achieve good TCP performance. This profoundly impacts TCP's congestion control mechanism. For example, consider Figure 5.6b. It is remarkable that throughput is almost the same at $d = 0$ ms and $d = 30$ ms, despite having 6% packet loss in the first case and less than 1% packet loss in the second.

Figure 5.7a depicts the congestion window over a 100 second interval during the $d = 0$ ms experiment.[7] Interestingly, the `cwnd` graph is far from the canonical sawtooth shape (e.g., Figure

[7] All congestion events in Figure 5.7a were fast retransmissions, except for one timeout at $t = 569$ s. `cwnd` is temporarily set to 1 MSS during fast retransmissions due to an artifact of FreeBSD's implementation of SACK recovery. For clarity, we cap `cwnd` at the size of the send buffer, and we remove fluctuations in `cwnd` which resulted from "bad retransmissions" that the FreeBSD implementation corrected in the course of its normal execution.



(a) TCP `cwnd` for $d = 0$, three hops.

(b) TCP loss recovery, three hops.

Figure 5.7: Congestion behavior of TCP over IEEE 802.15.4.

11(b) in [35]); `cwnd` is almost always maxed out even though losses are frequent (6%). This is specific to small buffers. In traditional environments, where links have higher throughput and buffers are large, it takes longer for `cwnd` to recover after packet loss, greatly limiting the sending rate with frequent packet losses. In contrast, **in LLNs, where send/receive buffers are small,** `cwnd` **recovers to the maximum size quickly after packet loss, making TCP performance robust to packet loss.**

Congestion behavior also provides insight into loss patterns, as shown in Figure 5.7b. Fast retransmissions (used for isolated losses) become less frequent as d increases, suggesting that they are primarily caused by hidden-terminal-related losses. Timeouts do not become less frequent as d is increased, suggesting that they are caused by something else.

5.7.4 Modeling TCP Goodput in an LLN

Our findings in Section 5.7.3 suggest that, in LLNs, `cwnd` is limited by the buffer size, not packet loss. To validate this, we analytically model TCP performance according to our observations in Section 5.7.3, and then check if the resulting model is consistent with the data. Comprehensive models of TCP, which take window size limitations into account, already exist [365]; in contrast, our model is *intentionally simple* to provide intuition.

Observations in Section 5.7.3 suggest that we can neglect the time it takes the congestion window to recover after packet loss. So, we model a TCP connection as *binary*: either it is sending data with a full window, or it is not sending new data because it is recovering from packet loss. According to this model, we can think of a TCP flow as a sequence of bursts. A *burst* is a sequence of full windows of data successfully transferred, which ends in a packet loss. After this loss, the flow spends some time recovering from the packet loss, which we call a *rest*. Then, the next burst begins. Burst lengths depend on the packet loss rate p and rest lengths depend on the RTT.

Let w be the size of TCP's flow window, measured in segments (for our experiments in Section 5.7.3, we would have $w = 4$). Define b as the average number of windows sent in a burst. The goodput of TCP is the number of bytes sent in each burst, which is $w \cdot b \cdot \text{MSS}$, divided by the duration of each burst. A burst lasts for the time to transmit b windows of data, plus the time to recover from the packet loss that ended the burst. The time to transmit b windows is $b \cdot \text{RTT}$. We define t_{rec} to be the time to recover from the packet loss. Then we have

$$B = \frac{w \cdot b \cdot \text{MSS}}{b \cdot \text{RTT} + t_{\text{rec}}}. \tag{5.1}$$

The value of b depends on the packet loss rate. We define a new variable, p_{win}, which denotes the probability that at least one packet in a window is lost. Then $b = \frac{1}{p_{\text{win}}}$.

To complete the model, we must estimate t_{rec} and p_{win}.

The value of t_{rec} depends on whether the retransmission timer expires (called an RTO) or a fast retransmission is performed. If an RTO occurs, the total time lost is the excess time budgeted to the retransmit timer beyond one RTT, plus the time to retransmit the lost segments. We denote the time budgeted to the retransmit timer as ETO. So the total time lost due to a timeout, assuming it takes about 2 RTTs to recover lost segments, would be $(\text{ETO} - \text{RTT}) + 2 \cdot \text{RTT} = \text{ETO} + \text{RTT}$. After a fast retransmission, TCP enters a "fast recovery" state [12, 217]. Fast recovery requires buffer space to be effective, however. In particular, if the buffer contains only four TCP segments, then the lost packet, and three packets afterward which resulted in duplicate ACKs, account for the entire send buffer; therefore, TCP cannot send new data during fast recovery, and instead stalls for one RTT, until the ACK for the fast retransmission is received. In contrast, choosing a larger send buffer will allow fast recovery to more effectively mask this loss [414].

As discussed in Section 5.7.3, these two types of losses may be caused by different factors. Therefore, we do not attempt to distinguish them on basis of probability. Instead, we use a very simple model: $t_{\text{rec}} = \ell \cdot \text{RTT}$. The constant ℓ can be chosen to describe the number of "productive" RTTs lost due to a packet loss. Based on the estimates above, choosing $\ell = 2$ seems reasonable for our experiments in Section 5.7 which used a buffer size of four segments.

To model p_{win}, we assume that, in each window, segment losses are independent. This gives us $p_{\text{win}} = 1 - (1-p)^w$, where p is the probability of an individual segment being lost (after link retries). Because p is likely to be small (less than 20%), we apply the approximation that $(1-x)^a \approx 1 - ax$ for small x. This gives us $p_{\text{win}} \approx wp$.

Applying these equations for t_{rec} and p_{win}, along with some minor algebraic manipulation to put our equation in a similar form to Equation 5.4, we obtain a model for TCP performance in LLNs, for small w and p.

$$B = \frac{\text{MSS}}{\text{RTT}} \cdot \frac{1}{\frac{1}{w} + \ell p} \tag{5.2}$$

Taking $\ell = 2$, as discussed above, we obtain the following model.

$$B = \frac{\text{MSS}}{\text{RTT}} \cdot \frac{1}{\frac{1}{w} + 2p} \tag{5.3}$$

where B, the TCP goodput, is written in terms of the maximum segment size MSS, round-trip time RTT, packet loss rate p ($0 < p < 1$), and window size w (sized to BDP, in packets). Figures 5.6a and 5.6b include the predicted goodput as dotted lines, calculated according to Equation 5.3 using the empirical RTT and segment loss rate for each experiment. **Our model of TCP goodput closely matches the empirical results.**

An established model of TCP outside of LLNs is the following [298, 336].

$$B = \frac{\text{MSS}}{\text{RTT}} \cdot \sqrt{\frac{3}{2p}} \tag{5.4}$$

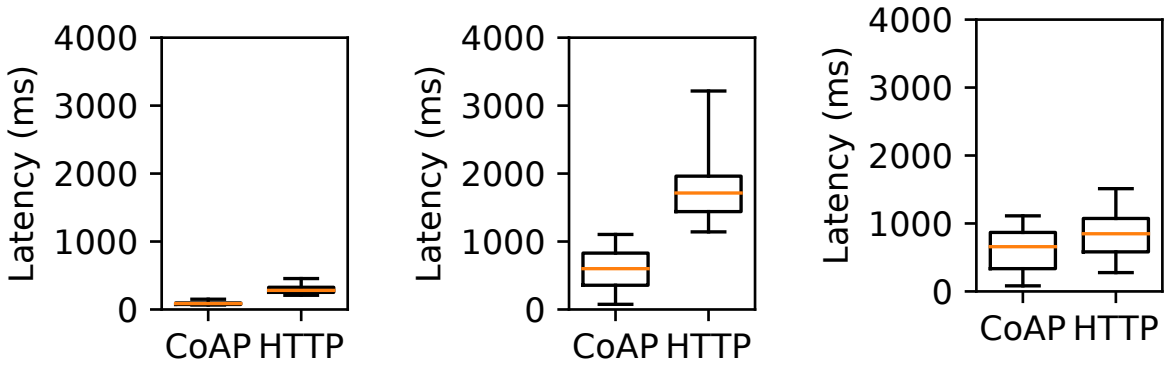
Equation 5.4 fundamentally relies on there being many competing flows, so we do not expect it to match our empirical results from Section 5.7.3. But, given that existing work examining TCP in LLNs makes use of this formula to ground new algorithms [238], the differences between Equations 5.3 and 5.4 are interesting to study. In particular, Equation 5.3 has an added $\frac{1}{w}$ in the denominator and depends on p rather than $\sqrt{p}$, explaining, mathematically, how TCP in LLNs is more robust to small amounts of packet loss. We hope that Equations 5.2 and 5.3 will provide a foundation for future research on TCP in LLNs.

5.8 TCP in LLN Applications

To demonstrate that TCP is practical for real IoT use cases, we compare its performance to that of CoAP, CoCoA, and unreliable UDP in three workloads inspired by real application scenarios: web server, sense-and-send, and event detection. We evaluate the protocols over multiple hops with duty-cycled radios and wireless interference, present in our testbed in the day (Section 5.4.2). In our experiments, nodes 12–15 (Figure 5.2) send data to a server running on Amazon EC2. The RTT from the border router to the server was $\approx$ 12 ms, much smaller than within the low-power mesh ($\approx$ 100-300 ms).

In our preliminary experiments, we found that in the presence of simultaneous TCP flows, tail drops at a relay node significantly impacted fairness. Implementing Random Early Detection (RED) [176] with Explicit Congestion Notification (ECN) support solved this problem. Therefore, we use RED and ECN for experiments in this section with multiple flows. While such solutions have sometimes been problematic since they are implemented in routers, they are more natural in LLNs because the intermediate "routers" relaying packets in an LLN typically also participate in the network as hosts.

We generally use a smaller MSS (3 frames) in this section, because it is more robust to interference in the day (Section 5.6). Furthermore, duty-cycling increases the RTT. It is natural to ask whether our conclusions in Section 5.7, including the model developed in Section 5.7.4, still hold in this setting. With a sleep interval of 100 ms, we qualitatively observed that, although `cwnd` tends to recover more slowly after loss, due to the smaller MSS and larger RTT, it is still "maxed out" past the BDP most of the time. Therefore, we expect our conclusion, that TCP is more resilient to packet loss, to also apply in this setting. One may consider adapting our model from Section 5.7.4



(a) No duty cycling.

(b) 1 s sleep interval.

(c) 1 s sleep interval with adaptive duty cycle.

Figure 5.8: Latency of web request: CoAP vs. HTTP/TCP.

to this setting by choosing a larger value of ℓ to reflect the fact that `cwnd` recovers from loss less quickly due to the smaller MSS. It is possible, however, that one could derive a better model by explicitly modeling the phase when `cwnd` is recovering, similar to other existing TCP models (in contrast to our model above, where we assume that the TCP flow is binary—either transmitting at a full window, or in backoff after loss). We leave a rigorous treatment of how these changes might affect the model, including an exploration of this idea, to future work.

Running TCP in these workloads motivates **Adaptive Duty Cycle** and **Finer-Grained Link Queue Management**, which we introduce below as they are needed.

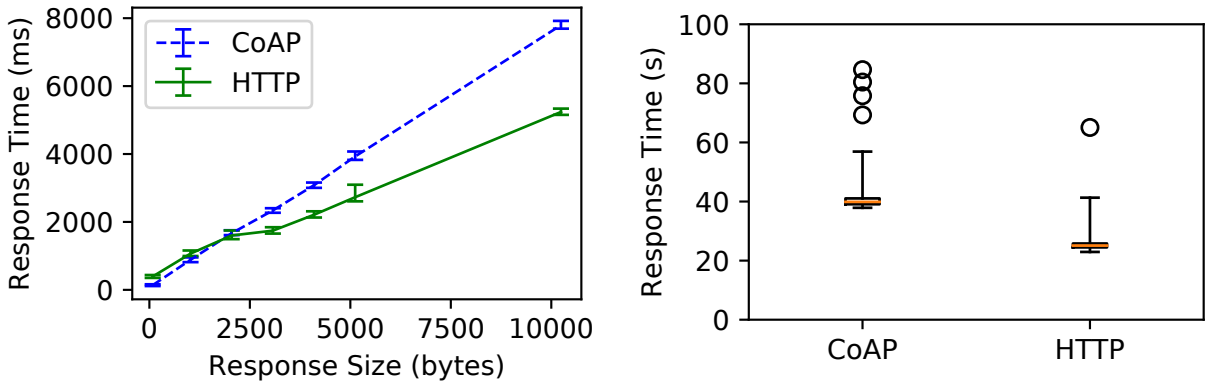
5.8.1 Web Server Application Scenario

To study TCP with multiple wireless hops and duty cycling, we begin with a web server hosted on a low-power device. We compare HTTP/TCP and CoAP/UDP (Section 5.4.1).

5.8.1.1 Latency Analysis

An HTTP request requires two round-trips: one to establish a TCP connection, and another for request/response. CoAP requires only one round trip (no connection establishment) and has smaller headers. Therefore, CoAP has a lower latency than HTTP/TCP when using an always-on link (Figure 5.8a). Even so, the latency of HTTP/TCP in this case is well below 1 second, not so large as to degrade user experience.

We now explore how a duty-cycled link affects the latency. Recall that leaf nodes in OpenThread (Section 5.4.1) periodically poll their parent to receive downstream packets, and keep their radios in a low-power sleep state between polls. We set the *sleep interval*—the time that a node waits between polls—to 1 s and show the latency in Figure 5.8b. Interestingly, HTTP's minimum observed latency is much higher than CoAP's, more than is explained by its additional round trip.



(a) Response time vs. size.

(b) 50 KiB response size.

Figure 5.9: Goodput: CoAP vs. HTTP/TCP.

Upon investigation, we found that this is because **the self-clocking nature of TCP [249] interacts poorly with the duty-cycled link**. Concretely, the web server receives the SYN packet when it polls its parent, and sends the SYN-ACK immediately afterward, at the *beginning* of the next sleep interval. The web server therefore waits for the *entire* sleep interval before polling its parent again to receive the HTTP request, thereby experiencing the worst-case latency for the second round trip. We also observed this problem for batch transfer over TCP; TCP's self-clocking behavior causes it to consistently experience the worst-case round-trip time.

To solve this problem, we propose a technique called **Adaptive Duty Cycling**. After the web server receives a SYN, it *reduces the sleep interval* in anticipation of receiving an HTTP request. After serving the request, it restores the sleep interval to its old value. Unlike early LLN link-layer protocols like S-MAC [499] that use an adaptive duty cycle, we use *transport-layer state* to inform the duty cycle. Figure 5.8c shows the latency with adaptive duty cycling, where the sleep interval is temporarily reduced to 100 ms after connection establishment. **With adaptive duty-cycling, the latency overhead of HTTP compared to CoAP is small, despite larger headers and an extra round trip for connection establishment.**

Adaptive duty cycling is also useful in high-throughput scenarios, and in situations with persistent TCP connections. We apply adaptive duty cycling to one such scenario in Section 5.8.2.

5.8.1.2 Throughput Analysis

In Section 5.8.1.1, the size of the web server's response was 82 bytes, intentionally small to focus on latency. In a real application, however, the response may be large (e.g., it may contain a batch of sensor readings). In this section, we explore larger response sizes. We use a short sleep interval of 100 ms. This is realistic because, using adaptive duty cycling, the sleep interval may be longer when the node is idle, and reduced to 100 ms only when transferring the response.

Figure 5.9a shows the total time from dispatching the request to receiving the full response, as we vary the size of the response. It plots the median time, with quartiles shown in error bars. HTTP takes longer than CoAP when the response size is small (consistent with Figure 5.8), but CoAP takes longer when the response size is larger. This indicates that while HTTP/TCP has a greater fixed-size overhead than CoAP (higher y-intercept), it transfers data at a higher throughput (lower slope). TCP achieves a higher throughput than CoAP because CoAP sends response segments one-at-a-time ("stop and wait"), whereas TCP allows multiple segments to be in flight simultaneously ("sliding window").

To quantify the difference in throughput, we compare TCP and CoAP when transferring 50 KiB of data in Figure 5.9b. **TCP achieves 40% higher throughput compared to CoAP, over multiple hops and a duty-cycled link.**

5.8.1.3 Power Consumption

TCP consumes more energy than CoAP due to the extra round-trip at the beginning. In practice, however, a web server is interactive, and therefore will be *idle* most of the time. Thus, the idle power consumption dominates. For example, TCP keeps the radio on 35% longer than CoAP for a response size of 1024 bytes, but if the user makes one request every 100 seconds on average, this difference drops to only 0.35%. Thus, we relegate in-depth power measurements to the sense-and-send application (Section 5.8.2), which is non-interactive.

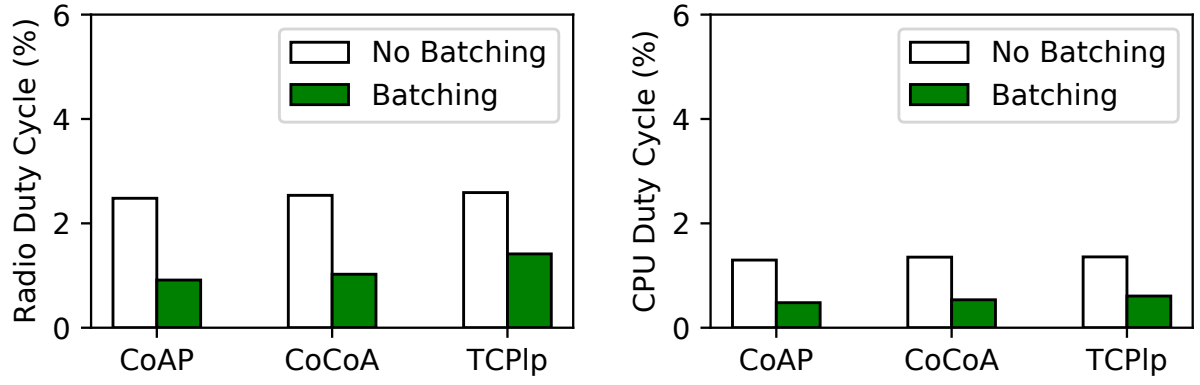
5.8.2 Sense-and-Send Application Scenario

We turn our focus to the common *sense-and-send* paradigm, in which devices periodically collect sensor readings and send them upstream. For concreteness, we model our experiments on the deployment of anemometers in a building, a real-world LLN use case described in Section 5.3.2. Anemometers collect measurements frequently (once per second), making heavy use of the transport protocol; given that our focus is on transport performance, this makes anemometers a good fit for our study. Other sensor deployments (e.g., temperature, humidity, building occupancy, etc.) sample data at a lower rate (e.g., 0.05 Hz), but are otherwise similar. Thus, *we expect our results to generalize to other sense-and-send applications.*

Nodes 12–15 (Figure 5.2) each generate one 82-byte reading every 1 second, and send it to the cloud server using either TCP or CoAP. We use most of the remaining RAM as an *application-layer queue* to prevent data from being lost if CoAP or TCP is in backoff after packet loss and cannot send out new data immediately. We make use of adaptive duty cycling for both TCP and CoAP, with a base sleep interval of four minutes (OpenThread's default) and decreasing it to 100 ms[8] when a TCP ACK or CoAP response is expected.

We measure a solution's *reliability* as the proportion of generated readings delivered to the server. Given that TCP and CoAP both guarantee reliability, a reliability measurement of less than 100% is caused by overflow of the application-layer queue due to poor network conditions

[8] 100 ms is comparable to ContikiMAC's default sleep interval of 125 ms.



(a) Radio duty cycle.

(b) CPU duty cycle.

Figure 5.10: Effect of batching on power consumption.

preventing data from being reliably communicated as fast as they are generated. Generating data more slowly would result in higher reliability.

5.8.2.1 Performance in Favorable Conditions

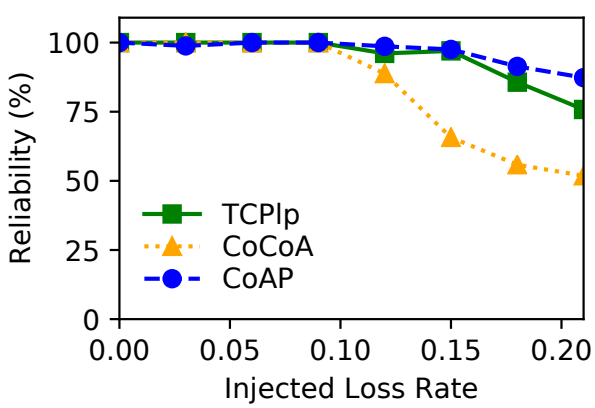
We begin with experiments in our testbed at night, when there is less wireless interference. We compare three setups: (1) CoAP, (2) CoCoA, and (3) *TCPlp*. We also compare two sending scenarios: (1) sending each sensor reading right away ("No Batching"), and (2) sending sensor readings in batches of 64 ("Batching") [280]. We ensure that packets in a CoAP batch are the same size as segments in TCP (five frames).

All setups achieved 100% reliability due to end-to-end acknowledgments (figures are omitted for brevity). Figures 5.10a and 5.10b also show that all the three protocols consume similar power; *TCP is comparable to LLN-specific solutions*.

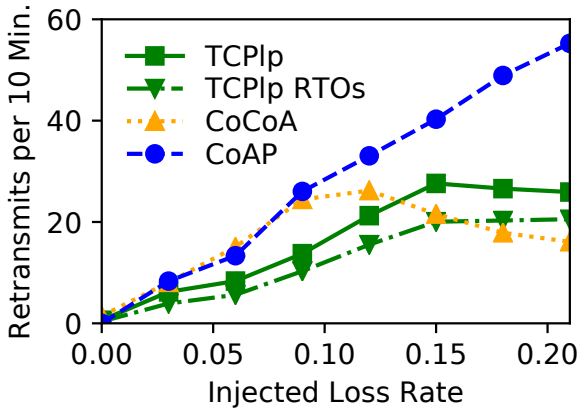
Both the radio and CPU duty cycle are significantly smaller with batching than without batching. By sending data in batches, nodes can amortize the cost of sending data and waiting for a response. Thus, batching is the more realistic workload, so we use it to continue our evaluation.

5.8.2.2 Resilience to Packet Loss

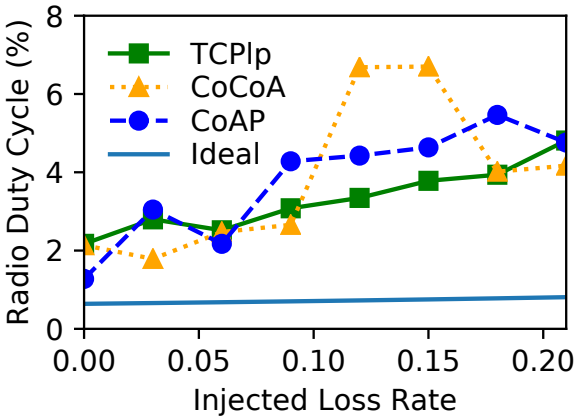
In this section, we inject uniformly random packet loss at the border router and measure each solution. The result is shown in Figure 5.11. Note that the injected loss rate corresponds to the *packet-level* loss rate *after* link retries and 6LoWPAN reassembly. Although we plot loss rates up to 21%, *we consider loss rates* $> 15\%$ *exceptional; we focus on the loss rate up to 15%*. A number of WSN studies have already achieved $> 90\%$ end-to-end packet delivery, using only link/routing layer techniques (not transport) [154, 275, 276]. In our testbed environment, we have not observed the loss rate exceed 15% for an extended time, even with wireless interference.



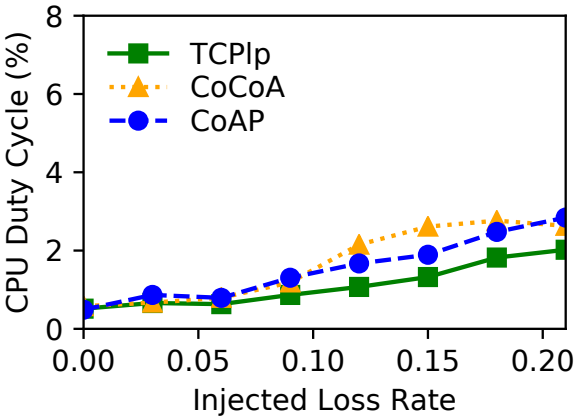
(a) Reliability.



(b) Transport-layer retries.



(c) Radio duty cycle.



(d) CPU duty cycle.

Figure 5.11: Performance with injected packet loss.

Both CoAP and TCP achieve nearly 100% reliability at packet loss rates less than 15%, as shown in Figure 5.11a. At loss rates greater than 9%, CoCoA performs poorly. The reason is that CoCoA attempts to measure RTT for retransmitted packets, and conservatively calculates the RTT relative to the first transmission. This results in an inflated RTT value that causes CoCoA to delay longer before retransmitting, causing the application-layer queue to overflow. Full-scale TCP is immune to this problem despite measuring the RTT, because the TCP timestamp option allows TCP to unambiguously determine the RTT even for retransmitted segments.

Figures 5.11c and 5.11d show that, overall, **TCP and CoAP perform comparably in terms of radio and CPU duty cycle**. At 0% injected loss, *TCPlp* has a slightly higher duty cycle, consistent with Figure 5.10. At moderate packet loss, *TCPlp* appears to have a slightly lower duty cycle. This may be due to TCP's sliding window, which allows it to tolerate some ACK losses without retries. Additionally, Figure 5.11b shows that, although most of TCP's retransmissions are explained by

timeouts, a significant portion were triggered in other ways (e.g., duplicate ACKs). In contrast, CoAP and CoCoA rely exclusively on timeouts, which has intrinsic limitations [512].

With exceptionally high packet loss rates (>15%), CoAP achieves higher reliability than TCP, because it "gives up" after just 4 retries; it exponentially increases the wait time between those retries, but then resets its RTO to 3 seconds when giving up and moving to the next packet. In contrast, TCP performs up to 12 retries with exponential backoff. Thus, TCP backs off further than CoAP upon consecutive packet losses, witnessed by the smaller retransmission count in Figure 5.11b, causing the application-layer queue to overflow more. This performance gap could be filled by parameter tuning.

We also consider an *ideal* "roofline" protocol to calculate a fairly loose lower bound on the duty cycle. This ideal protocol has the same header overhead as TCP, but learns which packets were lost for "free," without using ACKs or running MMC. Thus, it turns on its radio only to send out data and retransmit lost packets. The real protocols have much higher duty cycles than the ideal protocol would have (Figure 5.11c), suggesting that a significant amount of their overhead stems from determining which packets were lost—polling the parent node for downstream TCP ACKs/CoAP responses. This gap could be reduced by improving OpenThread's MMC protocol. For example, rather than using a fixed sleep interval of 100 ms when an ACK is expected, one could use exponential backoff to increase the sleep interval if an ACK is not quickly received. We leave exploring such ideas to future work.

5.8.2.3 Performance in Lossy Conditions

We compare the protocols over the course of a full day in our testbed, to study the impact of real wireless interference associated with human activity in an office. We focus on *TCPlp* and CoAP since they were the most promising protocols from the previous experiment. To ensure that *TCPlp* and CoAP are subject to similar interference patterns, we (1) run them simultaneously, and (2) hardcode adjacent *TCPlp* and CoAP nodes to have the same first hop in the multihop topology.

Improving Queue Management. OpenThread's queue management interacts poorly with TCP in the presence of interference. When a duty-cycled leaf node sends a data request message to its parent, it turns its radio on and listens until it receives a reply (called an "indirect message"). In OpenThread, the parent finishes sending its current frame (which may require link retries in the presence of interference), and then sends the indirect message. The duty-cycled leaf node keeps its radio on during this time, causing its radio duty cycle to increase. This is particularly bad for TCP, as its sliding window makes it more likely for the parent node to be in the middle of sending a frame when it receives a data request packet from a leaf node. Thus, **we modified OpenThread to allow indirect messages to preempt the current frame *in between link-layer retries***, to minimize the time that duty-cycled leaf nodes must wait for a reply with their radios on. Both TCP and CoAP benefited from this; TCP benefited more because it suffered more from the problem to begin with.

Power Consumption. To improve power consumption for both TCP and CoAP, we adjusted parameters according to the lossy environment: (1) we enabled link-layer retries for indirect messages, (2) we decreased the data request timeout and performed link-layer retries more rapidly for

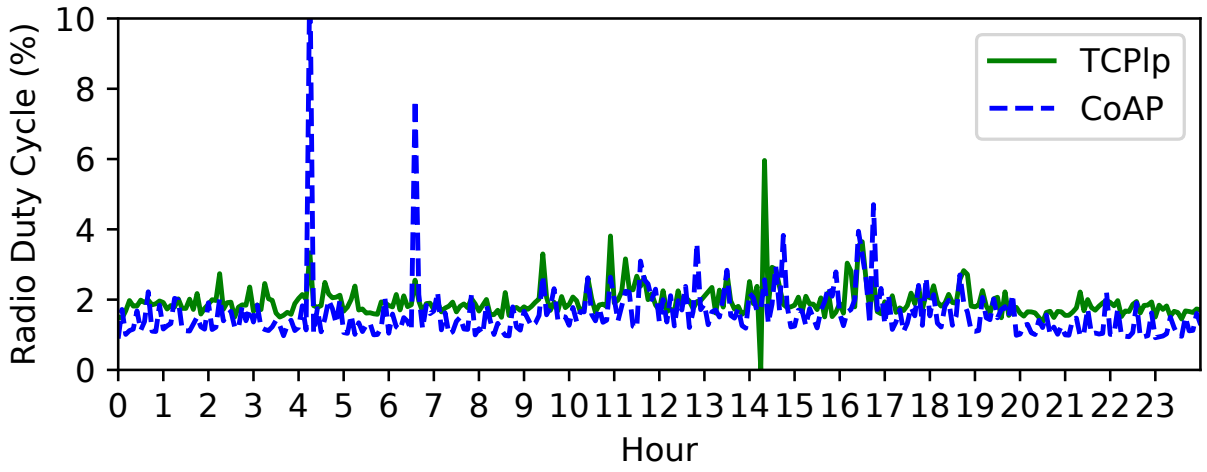


Figure 5.12: Radio duty cycle of TCP and CoAP in a lossy wireless environment, in one representative trial (losses are caused by natural human activity).

Protocol	Reliability	Radio Duty Cycle	CPU Duty Cycle
TCPlp	99.3%	2.29%	0.973%
CoAP	99.5%	1.84%	0.834%
Unreliable, no batching	93.4%	1.13%	0.52%
Unreliable, with batching	95.3%	0.734%	0.30%

Table 5.9: Performance in the testbed over a full day, averaged over multiple trials. The ideal protocol (Section 5.8.2.2) would have a radio duty cycle of ≈ 0.63%–0.70% under similarly lossy conditions.

indirect messages, to deliver them to leaves more quickly, and (3) given the high level of daytime interference, we decreased the MSS from five frames to three frames (as in Section 5.8).

Figure 5.12 depicts the radio duty cycle of TCP and CoAP for a trial representative of our overall results. **CoAP maintains a lower duty cycle than *TCPlp* outside of working hours, when there is less interference; *TCPlp* has a slightly lower duty cycle than CoAP during working hours, when there is more wireless interference.** *TCPlp*'s better performance at a higher loss rate is consistent with our results from Section 5.8.2.2. At a lower packet loss rate, TCP performs slightly worse than CoAP. This could be due to hidden terminal losses; more retries, on average, are required for indirect messages for TCP, causing leaf nodes to stay awake longer. Overall, CoAP and *TCPlp* perform similarly (Table 5.9).

5.8.2.4 Unreliable UDP

As a point of comparison, we repeat the sense-and-send experiment using a UDP-based protocol that *does not provide reliability*. Concretely, we run CoAP in "nonconfirmable" mode, in which it does not use transport-layer ACKs or retransmissions. The result is in the last two rows of Table 5.9. Compared to unreliable UDP, reliable approaches increase the radio/CPU duty cycle by 3×, in exchange for nearly 100% reliability. That said, the corresponding decrease in battery life will be *less* than 3×, because other sources of power consumption (reading from sensors, idle current) are also significant.

For other sense-and-send applications that sample at a lower rate, TCP and CoAP would see higher reliability (less application queue loss), but UDP would not similarly benefit (no application queue). Furthermore, the power consumption of TCP, CoAP, and unreliable UDP would all be closer together, given that the radio and CPU spend more time idle.

5.8.3 Event Detection Application Scenario

Finally, we consider an application scenario where multiple flows compete for available bandwidth in an LLN. One such scenario is event detection: sensors wait until an interesting event occurs, at which point they report data upstream at a high data rate. Because such events tend to be correlated, multiple sensors send data simultaneously.

Nodes 12-15 in our testbed simultaneously transmit data to the EC2 instance (Figure 5.2), which measures the goodput of each flow. We use the same duty-cycling policy as in Section 5.8.2. We divide each flow into 40-second intervals, measure the goodput in each interval, and compute the median and quartiles of goodput across all flows and intervals. The median gives a sense of aggregate goodput, and the quartiles gives a sense of fairness (quartiles close to the median are better).

Figure 5.13 shows the median and quartiles (as error bars) as the offered load increases. For small offered load, the per-flow goodput increases linearly. Once the aggregate load saturates the network, goodput declines slightly and the interquartile range increases, due to inefficiencies in independent flows competing for bandwidth. **Overall, TCP performs similarly to CoAP and CoCoA, indicating that TCP's congestion control remains effective despite our observations in Section 5.7.3 that it behaves differently in LLNs.**

5.9 Conclusion

TCP is the *de facto* reliability protocol in the Internet. Over the past 40 years, new physical-, datalink-, and application-layer protocols have evolved alongside TCP, and supporting good TCP performance was a consideration in their design. TCP is the obvious performance baseline for new transport-layer proposals. To warrant adoption, novel transports must be *much* better than TCP in the intended application domain.

In contrast, when LLN research flourished two decades ago, LLN hardware could not run full-scale TCP. The original system architecture for networked sensors [220], for example, targeted an

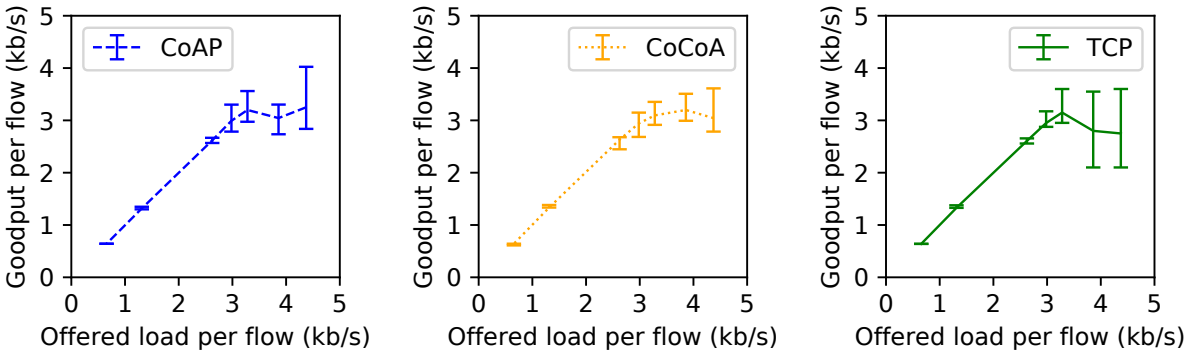


Figure 5.13: CoAP, CoCoA, and TCP with four competing flows.

8-bit MCU with only 512 *bytes* of memory. It naturally became taken for granted that TCP is too heavy for LLNs. Furthermore, contemporary research on TCP in WLANs [36, 33] suggested that TCP would perform poorly in LLNs even if the resource constraints were surmounted.

In revisiting the TCP question, after the resource constraints relaxed, we find that the expected pitfalls of wireless TCP actually do not carry over to LLNs. Although naïve TCP indeed performs poorly in LLNs, this is not due to fundamental problems with TCP as were observed in WLANs. Rather, it is caused by incompatibilities with a low-power link layer, which likely arose because canonical LLN protocols were developed in the absence of TCP considerations. We show how to fix these incompatibilities while preserving seamless interoperability with other TCP/IP networks. This enables a viable TCP-based transport architecture for LLNs.

5.9.1 Implications for Applications Relating to Cryptography

Applications based on expressive cryptography often incur networking overhead (Section 2.2.1.3). Networking overhead can be particularly significant when using an extremely constrained network like an LLN. For example, a protocol like JEDI, which we will explore in Chapter 7, is designed to be practical on LLN devices, but requires sending policy-based encryption ciphertexts, which are hundreds of bytes in size, over the network. Performant TCP, enabled by the work in this chapter, facilitates such applications by providing high throughput and reliability for transferring large cryptographic objects. In that sense, our work in this chapter is an application of the technique in Section 3.1.2—we provide a generic improvement to transport-layer networking in LLNs that facilitates applications based on expressive cryptography. Additionally, our work makes it easier to secure LLNs using widely used network security protocols like HTTPS and TLS, without relying on less widely used alternatives like DTLS.

5.9.2 Broader Implications for Networking

As our improvements to transport-layer networking in LLNs in this chapter are generic, they also benefit non-cryptographic applications. In particular, our results have several implications for LLNs moving forward. First, **the use of lightweight protocols that emulate part of TCP's functionality, like CoAP, needs to be reconsidered.** Protocol stacks like OpenThread should support full-scale TCP as an option. TCP should also serve as a benchmark to assess new LLN transport proposals.

Second, **full-scale TCP will influence the design of networked systems using LLNs.** Such systems are presently designed with application-layer gateways in mind (Section 5.3). Using TCP/IP in the LLN itself would allow the use of commodity network management tools, like firewalls and NIDS. TCP would also allow the application-layer gateway to be replaced with a network-layer router, allowing clients to interact with LLN applications in much the same way as a Wi-Fi router allows users to interact with web applications. This is much more flexible than the status quo, where each LLN application needs application-specific functionality to be installed at the gateway [502]. In cases where a new LLN transport protocol is truly necessary, the new protocol may be wise to consider the byte-stream *abstraction* of TCP. This would allow the application-layer gateway to be replaced by a *transport-layer gateway*. The mere presence of a transport layer, distinct from the application layer, goes a long way to providing interoperability with the rest of the Internet.

Third, **UDP-based protocols will still have a place in LLNs, just as they have a place in the Internet.** UDP is used for applications that benefit from greater control of segment transmission and loss response than TCP provides. These are typically real-time or multimedia applications where losing information is preferable to late delivery. It is entirely seemly for some sensing applications in LLNs, particularly those with similar real-time constraints, to transfer data using UDP-based protocols, even if TCP is an option. But TCP *still* benefits such applications by providing a reliable channel for control information. For example, TCP may be used for device configuration, or to provide a shell for debugging, without yet another reliability protocol.

In summary, LLN-class devices are ready to become first-class citizens of the Internet. To this end, we believe that TCP should have a place in the LLN architecture moving forward, and that it will help put the "I" in IoT for LLN-class devices.

Chapter 6

Using Cryptography Efficiently for Anonymous and Verifiable Data Sharing

This is the first of two chapters exploring the techniques in Section 3.2. We focus in this chapter on systems for storing and sharing data—systems in which users can store data, retrieve it later, and mark it as accessible to other users. A longstanding problem, in this context, is that of *secure storage*—providing useful security guarantees even when the storage server, and some users, are compromised by an adversary. To provide integrity guarantees when the server is compromised, it is natural to leverage a blockchain (Section 2.1.2.2). Unfortunately, blockchains are costly to use—they have high transaction latency and low transaction throughput (Section 2.2).

In this chapter, we design, implement, and evaluate Ghostor, a storage system that provides a notion of privacy that we call *anonymity* and a notion of integrity that we call *verifiable linearizability*, even when the storage server is compromised. By choosing anonymity as its privacy guarantee, Ghostor delinks user identities from data accesses while avoiding the need for expensive cryptographic tools that hide memory access patterns (e.g., multi-client ORAM). This is similar in spirit to the technique in Section 3.2.3. Ghostor leverages a blockchain (more generally, a ledger) to achieve its integrity guarantee, but we leverage the techniques from Section 3.2.1 and Section 3.2.2 to do so efficiently. These techniques (1) allow Ghostor use the blockchain rarely, so that the latency of issuing a blockchain transaction does not make any user-facing operation slow, and (2) make the frequency of blockchain transactions tunable based on the available transaction bandwidth and budget for issuing transactions, with a gradual degradation in security guarantee as blockchain transactions are less frequent. Ghostor incurs only a 4–5$\times$ overhead compared to an insecure baseline. Although significant, this cost may be worth it for security- and privacy-sensitive applications.

6.1 Introduction

Systems for remote data storage and sharing have seen widespread adoption over the past decade. Every major cloud provider offers it as a service (e.g., Amazon S3, Azure Blobs), and it is estimated

that 39% of corporate data uploaded to the cloud is related to file sharing [268]. Given the relentless attacks on servers storing data [237], the secure storage problem has seen much interest in both academia [315, 167, 429, 273, 262, 57, 190, 216, 304, 382] and industry [131, 453, 269, 384, 461].

Early systems addressing the secure storage problem [190, 261] have users encrypt and sign files. However, a sophisticated adversary can still:

- observe metadata about *users' identities* [110, 205, 245, 481]. Even if the files are encrypted, the adversary sees which users are sharing a file, which user is accessing a file at a given time, and the list of users in the system. Figure 6.1 shows an example where the attacker can conclude that Alice has cancer from such metadata. Further, this allows the attacker to learn the graph of user social relations [411, 437].

- perform active attacks. Despite the signatures, an adversary can revert a file to an earlier state as in a *rollback attack*, or hide users' updates from each other as in a *fork attack*, without being detected. These are dangerous if, for example, the shared file is Alice's medical profile, and she does not learn that her doctor changed her treatment.

Research over the past 15 years has striven to mitigate these attacks by providing *anonymity*—hiding users' identities from the storage server—or *verifiable consistency*—enabling users to detect rollback and fork attacks. In achieving these stronger security guarantees, however, state-of-the-art systems employ weaker threat models that rely on centralized trust: a trust assumption on a *few specific machines*. For example, they rely on a trusted party [440, 327], split the server into two components assuming one is honest [272, 364, 262], or assume the adversary is honest-but-curious (not malicious) [505, 89, 328, 31] meaning the attacker does not change the server's data or execution.

Attackers have notoriously performed highly targeted attacks, spreading malware with the ability to modify software, files, or source code [509, 508, 307]. In such attacks, a determined attacker can compromise any few *central servers*. Ideally, we would avoid *any trust* in the server or other clients, but unfortunately, that is impossible: Mazières and Shasha [337] proved that, if one cannot assume that clients are reliably online [273], clients cannot detect fork attacks without placing some trust in the server. Tools from expressive cryptography, such as multi-client ORAM and blockchains, provide an avenue to achieving strong privacy and integrity guarantees without central trust, but they are expensive to use. Hence, this chapter asks the question: **Can we achieve strong privacy and integrity guarantees in a data-sharing system with *practical overhead*, without relying on *centralized trust*?**

To answer this question, we design and build Ghostor, an object store based on *decentralized trust* that achieves *anonymity* and *verifiable linearizability* (abbreviated VerLinear). At a high level, anonymity[1] means that the protocol does not reveal directly to the server any user identity with any request, as previously defined in the secure storage literature [505, 272, 364, 328]. As shown in Figure 6.1, the server does not see which user owns which objects, which users have read or write

[1]Outside of secure storage, *anonymity* is sometimes defined differently. In secure messaging, for example, an anonymous system is expected to hide the timing of accesses [222] and which files/mailboxes are accessed, but not necessarily the system's membership [128].

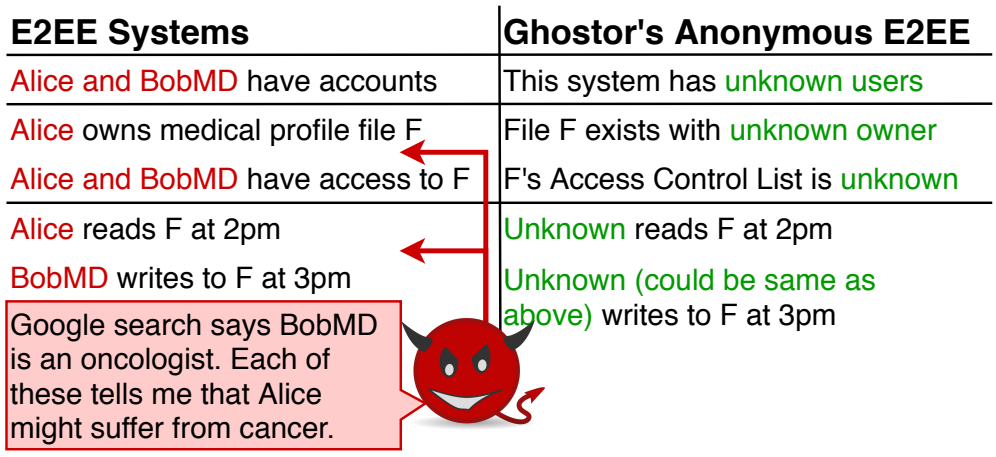


Figure 6.1: An example of what a server attacker sees in a typical end-to-end encrypted (E2EE) system versus Ghostor's Anonymous E2EE.

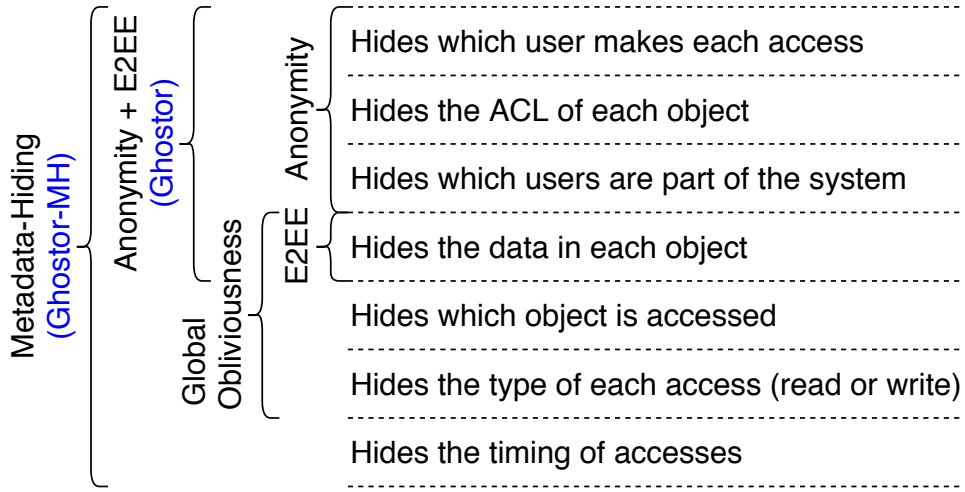


Figure 6.2: Information leakage in a data-sharing system and associated privacy properties.

permissions to a given object, or even who are the users of the system. The server essentially sees **ghost**s accessing the **stor**age, hence the name "**Ghostor**." VerLinear means clients can verify that each write is reflected in later reads, except for benign reordering of concurrent operations as formalized by linearizability [218]. To achieve these properties, we build Ghostor's integrity on top of a consistent storage primitive based on decentralized trust, like a blockchain [353, 93, 506] or verifiable ledger [235, 160], while using it only *rarely* (technique from Section 3.2.1).

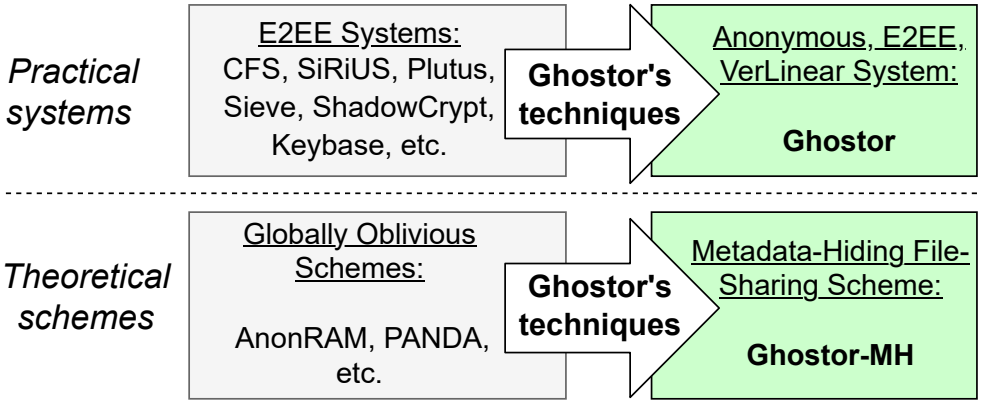


Figure 6.3: Ghostor's contributions. Ghostor's techniques can be applied to both oblivious and non-oblivious systems.

6.1.1 Hiding User Identities

Achieving anonymity in practical data-sharing systems like Ghostor is difficult because common system design paradigms, like user login, per-user mailboxes on the server, and client-side caching, let the server track users. As we explain in Section 6.4, even using user-specific keys to sign updates to data objects can reveal to the server which user performed the update, and requires knowledge of the ACL to check that the signer is an authorized user. We re-architect the system to avoid these paradigms (Section 6.4), using data-centric key distribution and encrypted key lists instead of server-side ACLs. Like prior systems [181, 4, 296], Ghostor uses cryptographic keys as capabilities, allowing the server and other users to verify that each access is performed by an authorized user. Ghostor also leverages this technique to achieve anonymity by having all users authorized to perform a particular operation on an object (e.g., all users with read access to an object) *share* the same capability for performing that operation on that object, and by distributing these capabilities to users without revealing ACLs to the server. We find this technique, *anonymously distributed shared capabilities*, interesting because anonymity is not typically a goal of public-key access control [181, 4] or capability-based systems [312, 423, 343].

An additional challenge is to guard against resource abuse while preserving anonymity. This is typically done by enforcing per-user resource quotas (e.g., Google Drive requires users to pay for additional space), but this is incompatible with Ghostor's anonymity. One solution is for users to pay for each operation via an anonymous cryptocurrency (e.g., Zcash [506]), but this puts an expensive blockchain operation in the critical path. To avoid this, Ghostor leverages blind signatures [95, 105, 104] to allow a user to pay the Ghostor server for service in bulk and in advance, while removing the linkage between payments and operations.

Relationship to obliviousness. Figure 6.2 positions Ghostor's anonymity with respect to other privacy properties. Global obliviousness [31, 327], which hides which *object* is accessed across all uncompromised objects and users in the system, is orthogonal to Ghostor's anonymity, which

hides which *user* performs each access. Obliviousness and anonymity are also complementary: (1) In some cases, without obliviousness, users may be identified based on access patterns. (2) Without anonymity, knowing which user issued a request may reveal information about what data that request may access. Ghostor's techniques for anonymity are a *transformation* (Figure 6.3):

- If applied to an E2EE system, we obtain **Ghostor, an anonymous E2EE system**.
- If applied to a globally oblivious scheme, we obtain **Ghostor-MH, a data-sharing scheme that hides all metadata**.

Hiding metadata from a malicious adversary, as in Ghostor-MH, is a very strong guarantee—existing globally oblivious schemes inherently reveal user identities [327] or assume the adversary is honest-but-curious [328, 31]. However, globally oblivious data-sharing schemes, like Ghostor-MH, are theoretical schemes that are far from practical. Thus, Ghostor-MH is only a proof of concept demonstrating the power of Ghostor's techniques to lift a globally oblivious scheme all the way to virtually zero leakage for a malicious adversary. As such, we do not include a full description of Ghostor-MH in this dissertation; we relegate it to our full Ghostor paper [227].

An important similarity between anonymity and global obliviousness, in the context of Ghostor, is that both prevent the adversary from being able to link user identities to which data items are accessed. Anonymity achieves this by preventing the adversary from learning user identities, and global obliviousness achieves this by preventing the adversary from learning which data items are accessed. Given this similarity, Ghostor's privacy design is in the spirit of the technique in Section 3.2.3. Specifically, anonymity is a cheaper way than global obliviousness to delink user identities from accessed data in Ghostor's setting, leading us to design Ghostor to provide anonymity but not global obliviousness. That said, we do not consider this to be an application of the technique in Section 3.2.3 because it affects Ghostor's security guarantees—as described above, anonymity and global obliviousness are different from one another.

6.1.2 Verifiable Consistency

To provide VerLinear, prior work either has clients sign hashes [273] so that clients can verify that they see the same hash, or store hashes on a separate hash server [262], trusted not to collude with the storage server. Neither technique can be used in Ghostor: client signatures are at odds with anonymity, and the hash server is a trusted party, which Ghostor aims to avoid.

One way to adapt the prior designs to Ghostor's decentralized trust is to store hashes on a blockchain, which can be accomplished by running the hash server in a smart contract. Unfortunately, this design is **too slow to be practical**. The client posts a hash on the blockchain for every object write, which is expensive: blockchains incur high latency per transaction, have low transaction throughput, and require cryptocurrency payment for each transaction [93, 353, 506].

To sidestep the limitations of a blockchain, we design Ghostor to only use the blockchain rarely and outside of the critical path. Ghostor divides time into intervals called *epochs*. At the end of each epoch, the Ghostor server publishes to the blockchain a small *checkpoint*, which summarizes the operations performed during that epoch for all objects and users in the system. Each user can

Goal	Technique
Anonymous user access control	Anonymously distributed shared capabilities (Section 6.4)
Anonymous server integrity verification	Verifiable anonymous history (Section 6.5)
Concurrent operations on a single object	Optimized `GET`s, two-phase protocol for `PUT`s (Section 6.5.4)
Anonymous resource abuse prevention	Blind signatures and proof of work (Section 6.6)
Hiding user IP addresses	Anonymous network, e.g., Tor (Section 6.9)

Table 6.1: Our goals and how Ghostor achieves each one.

then verify that the results of their accesses during the epoch are consistent with the checkpoint. The consistency properties of a blockchain ensure all clients see the same checkpoint, so the server is committed to a single history of operations and cannot perform a fork attack. The epoch time is a tunable system parameter; increasing it decreases the frequency of blockchain operations, reducing the cost of the system, but also increases the time that clients must wait before running the verification procedure. This gives flexibility to strike the right balance between cost and security for each application or deployment scenario (technique in Section 3.2.2). Commit chains [271] and monitoring schemes [73, 451] are based on similar checkpoints, but Ghostor applies them to object storage while maintaining users' anonymity.

A significant obstacle is that a hash-chain-based history is not amenable to concurrent appends. Each entry in the history contains the hash of the previous entry, causing one operation to fail if a concurrent operation appends a new entry. Existing techniques for concurrent operations, such as SUNDR's VSLs [315], reveal *per-user* version numbers that would undermine Ghostor's anonymity. Our insight in Ghostor is to have the *server*, not the client, populate the hash of the previous entry when appending a new entry. To make this safe despite a malicious adversary, we carefully design a conflict resolution strategy, involving multiple *linked* entries in the history for each write, that prevents attackers from manipulating data via replay or time-stretch attacks.

We call the resulting design a *verifiable anonymous history*.

6.1.3 Summary of Contributions

Our goals and techniques are summarized in Table 6.1. Overall, this chapter's contributions are:

- We design an object store providing anonymity and verifiable linearizability based only on *decentralized trust*.
- We develop techniques to (1) share capabilities for anonymity and distribute them anonymously, (2) create and checkpoint a verifiable anonymous history, and (3) support concurrent operations on a single object with a hash-chain-based history.

- We combine these with existing building blocks to instantiate Ghostor, an object store with anonymity and VerLinear.
- We also apply these to a globally oblivious scheme to instantiate Ghostor-MH, which hides nearly all metadata.

We also implemented Ghostor and evaluated it on Amazon EC2. Overall, Ghostor brings a 4–5× throughput overhead on top of a simplistic and completely insecure baseline. There are two types of latency overhead. Completing an individual operation takes several seconds. Afterward, it may take several minutes for a checkpoint to be incorporated into the blockchain, to confirm that no active attack has occurred for a batch of operations. We explain how these latencies play out in the context of a particular application, EHR Sharing (Section 6.8).

6.2 System Overview

Ghostor is an object store, which stores unstructured data items ("objects") and allows shared access to them by multiple users. We instantiate Ghostor as an object store (as in Amazon S3 or Azure Blobs) because it is a basic primitive on top of which more complex systems can be built. Figure 6.4 illustrates Ghostor's architecture. Multiple users, with separate clients, have shared access to objects on the Ghostor server.

Server. The Ghostor storage server processes requests from clients. At the end of each epoch, the server generates a single small checkpoint and publishes it to the blockchain.

Client. The client software consists of a Ghostor library, linked into applications, and a verification daemon, which runs as a separate process. The Ghostor library receives requests from the application and interacts with the server to satisfy each request. Upon accessing an object, the library forwards a digest summarizing the operation to the verification daemon. At the end of each epoch, the daemon (1) fetches object histories from the server, (2) verifies that they are consistent with the server's checkpoint on the blockchain, and (3) checks that the digests collected during the epoch are consistent with the object histories, as explained in Section 6.5.

The daemon stores the user's keypair. If a user loses her secret key, she loses access to all objects that she created or was granted access to. Similarly, an attacker who steals a user's secret key can impersonate that user. To securely back up her key on multiple devices, a user can use standard techniques like secret sharing [421, 468, 412]. A user who accesses Ghostor from multiple devices uses the same key on all devices.

Application developers interact with Ghostor using the API below. Developers can work with usernames, ACLs, and object IDs, but Ghostor clients will not expose them to the Ghostor server. Below is a high-level description of each API call; a step-by-step technical description is in Section 6.7.

♢ **create_user**(): Creates a Ghostor user by generating keys for a new user. This operation runs entirely in the Ghostor client—the server does not know this operation was invoked.

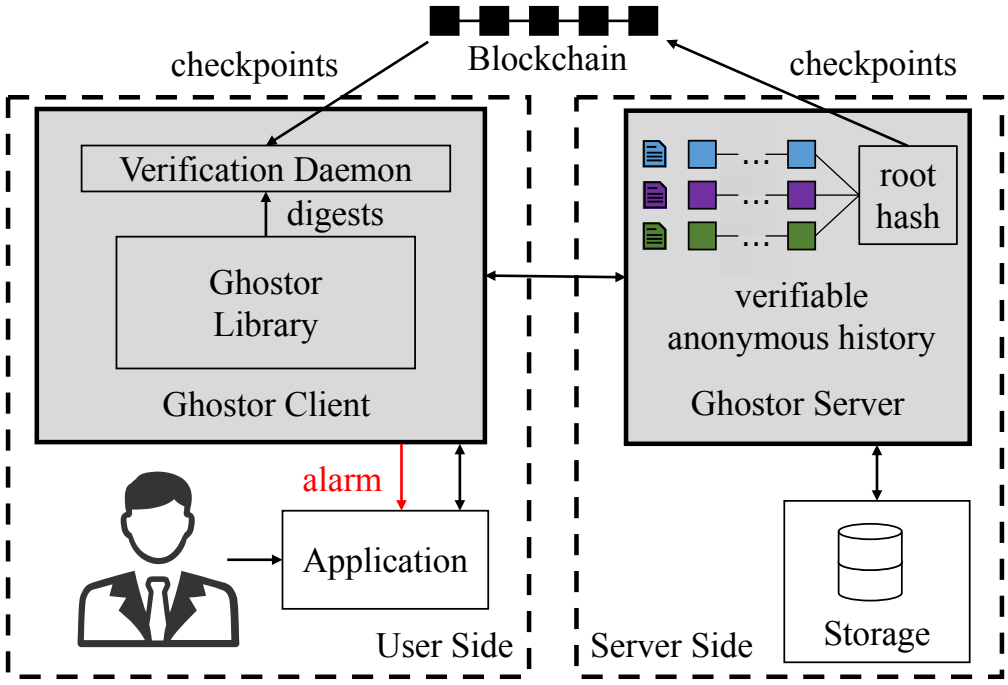


Figure 6.4: System overview of Ghostor. Shaded areas indicate components introduced by Ghostor.

◊ user.**pay**(*sum*): Users pay the server through an anonymous cryptocurrency such as Zcash [506], and obtain *tokens* from the server proportional to the amount paid. These tokens can later be *anonymously redeemed* and used as proof of payment when invoking the below API functions.
◊ user.**create_object**(*id*): Creates an object with ID *id*, owned by user who invokes this. The client expends one token obtained from a previous call to **pay**. The *id* can be a meaningful name (e.g., a file path). It lives only within the client—the server receives some cryptographic identifier—so different clients can assign different *id*s to the same object.
◊ user.**set_acl**(*id*, *acl*): The user who invokes this must be the owner of the object with ID *id*. This function sets a new ACL for that object. For simplicity, only the owner of an object can set its ACL, but Ghostor can be extended to permit other users as well. The client encodes *acl* into an object header that hides user identities, as in Section 6.4. If new users are given access, they are notified via an out-of-band channel. Existing data-sharing systems also have this requirement; for example, Dropbox and Box send an email with an access URL to the user. In Ghostor, all keys are transferred in-band; the out-of-band channel is used only to *inform* the user that she has been given access. Ghostor does not require a specific out-of-band channel; for example, one could use Tor [144] or secure messaging [455, 222].
◊ user.**get_object**(*id*), user.**put_object**(*id*, *content*): The user can `GET` or `PUT` an object if permitted by its ACL.

6.3 Threat Model and Security Guarantees

Against a malicious attacker who has compromised the server, Ghostor provides:

- verifiable linearizability, as described in Section 6.3.2, and
- a notion of user anonymity, described in Section 6.3.3: briefly, it does not reveal user identities, but reveals object access patterns. Ghostor-MH additionally hides access patterns.

Ghostor does not protect against attacks to availability. Nevertheless, its anonymity makes it more difficult for the server to selectively deny service to (or fork views of) certain users. Users, and the Ghostor client instances running on their behalf, can be malicious and can collude with the server.

Formal definitions and proofs for these properties require a large amount of space, so we relegate them to Appendix A. In this section, we include only *informal* definitions.

6.3.1 Assumptions

Ghostor is designed to derive its security from decentralized trust. Thus, our threat model assumes an adversary who can compromise any few machines, as described below.

Blockchain. Ghostor makes the standard assumption that the blockchain is immutable and consistent (all users see the same transaction history). This is based on the assumption that, in order to attack a blockchain, the adversary cannot simply compromise a few machines, but rather a significant fraction of the world's computing power. Ghostor's design is not tied to a specific blockchain. Our implementation uses Zcash [506] because it supports both public and private transactions; we use Zcash's private transactions for Ghostor's anonymous payments. The privacy guarantees of Zcash can be implemented on top of other blockchains as well [47].

Network. We assume clients communicate with the server in a way that does not reveal their network information. This can be done using mixnets [107] or secure messaging [455, 222] based on decentralized trust. Our implementation uses Tor [144].

6.3.2 Verifiable Linearizability

If an attack is immediately detectable to a user—for example, if the server fails to honor payment or provides a malformed response (e.g., bad signature)—we consider it an attack on *availability*, which Ghostor does not prevent.

Clients should be able to detect active attacks, including fork and rollback attacks. Some reordering of concurrent operations, however, is benign. We use *linearizability* [218] to define when reordering at the server is considered benign or malicious. *Informally*, linearizability requires that after a `PUT` completes, all later `GET`s return the value of either (1) that `PUT`, (2) a `PUT` that was concurrent with it, or (3) a `PUT` that comes after it. We provide a more formal definition in Appendix A.2. Ghostor provides *verifiable linearizability* (abbreviated *VerLinear*). This means that if the server deviates from linearizability, clients can detect it at the end of the epoch. We

discuss how to choose the epoch length in Section 6.10. Ghostor does not provide consistency guarantees for malicious user, or for objects for which a malicious user has write access.

Guarantee 1 (Verifiable linearizability)**.** *For any object F and any list E of consecutive epochs, suppose that, for each epoch in E, the set of honest users who ran the verification procedure includes all writers of F in that epoch (or is nonempty if F was not written).* ***If*** *the server did not linearizably execute the operations that verifying clients performed in the epochs that they verified,* ***then*** *at least one of the verifying clients will encounter an error in the verification procedure and can generate a proof that the server misbehaved.*

6.3.3 Anonymity

As explained in Section 6.1.1, Ghostor's anonymity means that the server sees no user identities associated with any action. In particular, an adversary controlling the server cannot tell which user accesses each object, which users are authorized to access each object, or which users are part of the system.

We informally define Ghostor's privacy via a *leakage function*: what the server learns when a user makes each API call (Section 6.2). For **create_object** – **put_object**, the server learns the object ID, the type of operation, and whether the user is authorized according to the object's ACL (past and present). The server also sees the time of the operation, and the size of the encrypted ACL and encrypted object, which can be hidden via padding at an extra cost. **create_user** leaks no information to the server, and **pay** reveals the sum paid and when. The server learns no user identities, no object contents, and no ACLs. If the attacker has compromised some users, he learns the contents of objects those users can access, including prior versions encrypted under the same key. Collectively, the verification daemons leak the number of clients performing verification for each object. If all clients in an object's ACL are honest and running, this equals the ACL size. If the ACL is padded to a maximum size, the owner should run verification more times to hide the ACL size. Ghostor does *not* hide access patterns or timing (Figure 6.2). An adversary who uses this information cannot see the contents of files and ACLs because they are encrypted. But such an adversary could try to deduce correlations between which users issue different operations based on access patterns and timing, and in some cases, identify the user based on that information. This can be partially mitigated by carefully designing the application using Ghostor (Section 6.4.5). In contrast, Ghostor-MH does hide access patterns. In Appendix A.1, we formally define Ghostor's privacy guarantee in the simulation paradigm of SMPC.

6.4 Hiding User Identities

System design paradigms used in typical data-sharing systems are incompatible with anonymity. We identify the incompatible system design patterns and show how Ghostor replaces them. Ultimately, we arrive at *anonymously distributed shared capabilities*, which allow Ghostor to enforce access control for anonymous users without server-visible ACLs.

Keypair or Key	Description
(PVK, PSK)	Signing keypair used to set ACL
(RVK, RSK)	Signing keypair used to get object
(WVK, WSK)	Signing keypair used to put object
(OSK)	Symmetric key for object contents

Table 6.2: Per-object keys in Ghostor. The server uses the global signing keypair $(\mathsf{SVK}, \mathsf{SSK})$ to sign digests for objects.

6.4.1 No User Login or User-Specific Mailboxes

Data-sharing systems typically have some storage space on the server, called an *account file*, dedicated to a user's account. For example, Keybase [269] has a user account and Mylar [382] has a user mailbox where the user receives a key to a new file. Accesses to the account file, however, can be used to *link user operations*. As an example, suppose that when a user accesses an object, her client first retrieves the decryption key from a user-specific mailbox. This violates anonymity because the server can tell whether or not two accesses were made by the same user, based on whether the same mailbox was accessed first. Instead, Ghostor's anonymity requires that *any sequence of API calls (Section 6.2) with the same inputs, when performed by any honest user, results in the same server-side accesses*.

Ghostor does not have any user-specific storage as in existing systems. To allow in-band key exchange, Ghostor associates a *header* with each object. The object header functions like an object-specific mailbox, in that it is used to distribute the object's keys among users who have access to the object. Unlike a user-specific mailbox, it preserves anonymity because, for a given object, each user reads the *same* header before accessing it.

6.4.2 No Server-Visible ACLs

An honest server must be able to prevent unauthorized users from modifying objects, and users must be able to verify that objects returned by the server were produced by authorized writers. This is typically accomplished by having writers sign objects, and having the server check that the user who signed the object is on the object's ACL. However, this requires the ACL to be visible to the server, which violates anonymity.

We observe that by switching to a design based on *shared capabilities*, we can allow the server and other users to verify that writes are indeed made by authorized users, without requiring the server or other users to know the ACL of the object, or which users are authorized. Every Ghostor object has three associated signing keypairs (Table 6.2). All users of the object (and the server)

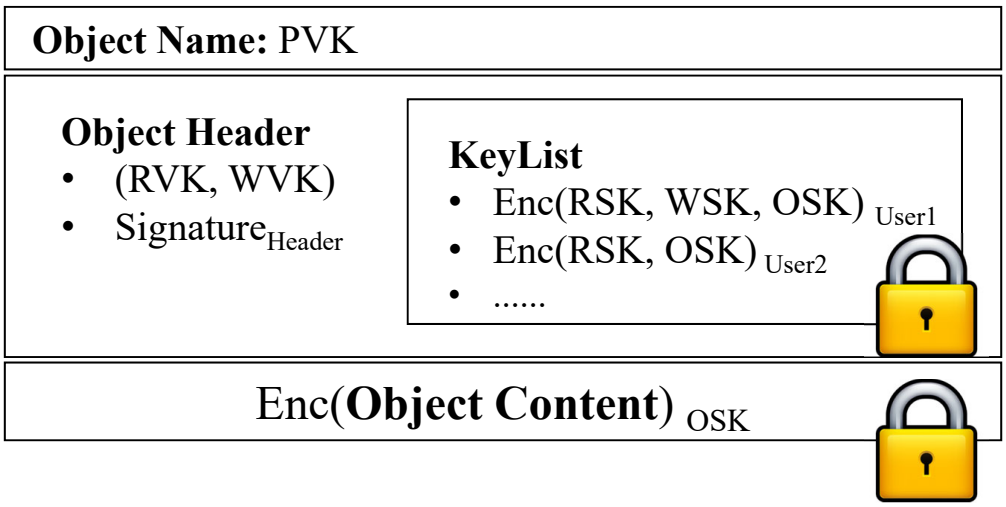


Figure 6.5: Object layout in Ghostor.

know the verifying keys PVK, RVK, and WVK because PVK is the name of the object, and RVK and WVK are in the object header; the associated signing keys PSK, RSK, and WSK are *capabilities* that grant access to set the ACL, get the object, and put the object, respectively. To distribute these capabilities to users in the object's ACL, the owner places a *key list* in the object header. The key list contains, for each user in the ACL, a list of capabilities encrypted under that user's public key. The owner randomly shuffles the key list and, optionally, pads it to a maximum size to hide each user's position. If a user has read/write access to an object, her entry in the key list contains WSK, RSK, and OSK; a user with only read access is given a dummy key instead of WSK. Crucially, different users with the same permission *share the same capability*, so the server cannot distinguish between users on the basis of which capability they use. When accessing an object, a user downloads the header and decrypts her entry in the key list to obtain OSK (used to decrypt the object contents) and her capabilities for the object.

Users sign updates to the object with WSK, allowing the server and other users to verify that each update is made by a user with write access. PSK is stored locally by the owner and is used to sign the header. The owner can set the object's ACL by (1) freshly sampling (RVK, RSK), (WVK, WSK), and OSK, (2) re-encrypting the object with OSK and signing it with WSK, (3) creating a new object header with an updated key list, (4) signing the new header with PSK, and (5) uploading it to the server. (RVK, RSK) will be relevant in Section 6.5.

Ghostor's object layout is summarized in Figure 6.5.

6.4.3 No Server-Visible User Public Keys

Prior systems [315] reveal the user's public key to the server when the client interacts with it. For example, SUNDR requires users to provide a signature along with each operation. First, the signature itself could leak the user's public key. Second, to check the legitimacy of writes, the

server needs to know the user's public key to verify the signature. The server can use the public key as a *pseudonym* to track users.

The key list in Section 6.4.2, however, potentially leaks users' public keys: each entry in the key list is a set of capabilities encrypted under a user's public key, but public-key encryption is only guaranteed to hide the message being encrypted, not the public key used to encrypt it. For example, an RSA ciphertext leaks which public key was used for encryption. Therefore, Ghostor uses *key-private* encryption [44], which is guaranteed to hide both the message and the public key.

In summary, Ghostor has users *share* capabilities for anonymity, and then distributes the capabilities anonymously, without revealing ACLs to the server. We call the resulting technique *anonymously distributed shared capabilities*.

6.4.4 No Client-Side Caching

Assuming that an object's ACL changes rarely, it may seem natural for clients to locally cache an object's keypairs $(\mathsf{RVK}, \mathsf{RSK})$ and $(\mathsf{WVK}, \mathsf{WSK})$, to avoid downloading the header on future accesses to that object. Unfortunately, the mere fact that a client did not download the header before performing an operation tells the server that the *same user* recently accessed that object. As a result, Ghostor's anonymity prohibits user-specific caching. That said, *server-side* caching of commonly accessed objects is allowed.

6.4.5 Careful Application Design

Ghostor does not hide access patterns or timing information from the server. A sophisticated adversary could, for example, deny or delay accesses to a particular object and see how access patterns shift, to try and deduce which user made which accesses. Therefore, one should carefully design the application using Ghostor to avoid leaking user identities in its access patterns. For example, just as Ghostor has no client-side caching or user-specific mailboxes, an application using Ghostor should avoid caching data locally to avoid requests to the server or using an object as a user-specific mailbox. Note that Ghostor-MH hides these access patterns.

6.5 Achieving Verifiable Consistency

Ghostor's *verifiable anonymous history* achieves the "verifiable equivalent" of a blockchain for critical-path operations, while using the underlying blockchain rarely. It consists of: (1) a hash chain of digests, (2) periodic checkpoints on a real blockchain, and (3) a verification procedure that does not require knowledge of user identities.

6.5.1 Hash Chain of Digests

We now achieve fork consistency for a single object in Ghostor using techniques inspired from SUNDR [315], but modified because SUNDR is not anonymous. Each access to an object, whether

Field	Description
Epoch	epoch when operation was committed
PVK, WVK, RVK	permission/writer/reader verifying key
$\mathsf{Hash}_{\mathsf{prev}}$	hash of previous digest in chain
$\mathsf{Hash}_{\mathsf{keylist}}$	hash of key list
$\mathsf{Hash}_{\mathsf{data}}$	hash of object contents
$\mathsf{Sig}_{\mathsf{client}}$	client signature with RSK, WSK, or PSK
$\mathsf{Sig}_{\mathsf{server}}$	server signature using SSK
nonce	random nonce chosen by client

Table 6.3: A digest for an operation in Ghostor.

a `GET` or a `PUT`, is summarized by a *digest* shown in Table 6.3. The object's history is stored as a chain of digests.

To access the object, a client first produces a digest summarizing that operation as in Table 6.3. This requires fetching the object header from the server, so that the client can obtain the secret key (RSK, WSK, or PSK) for the desired operation. Then the client fetches the latest digest for the object and computes $\mathsf{Hash}_{\mathsf{prev}}$ in the new digest. To `GET` the object, the client copies $\mathsf{Hash}_{\mathsf{data}}$ from the latest digest; to `PUT` it, the client hashes the new contents to obtain $\mathsf{Hash}_{\mathsf{data}}$. If the client is changing permissions, then $\mathsf{Hash}_{\mathsf{keylist}}$ is calculated from the new header; otherwise, it is copied from the latest digest.

Then the client signs the digest with the appropriate key and provides the signed digest to the server. The server signs the digest using SSK, appends it to a log, and returns the signed digest and the result of the operation. At the end of the epoch, the client downloads the digest chain for that object and epoch, and verifies that (1) it is a valid history for the object, and that (2) it contains the operations performed by that client. We specify protocol details in Section 6.7.

Ghostor's digests differ from SUNDR in two main ways. First, for anonymity, a client does not sign digests using the user's secret key, but instead uses RSK, WSK, or PSK, which can be verified without knowing the user's public key. When inspecting the digest, the server no longer learns which user performed the operation, only that the user has the required permission. Second, each digest is signed by the server. Thus, if the server violates linearizability, the client can assemble the offending digests into a *proof of misbehavior*.

6.5.2 Checkpoint and Verification

The construction so far is susceptible to fork attacks [315], in which the server presents two users with different views over the same object. To detect fork attacks, Ghostor requires the server to produce a *checkpoint* at the end of each epoch, consisting of the hash of the object's latest digest and the epoch number, and publish the checkpoint to the blockchain. The *verification procedure* run by a client consists of fetching the checkpoint from the blockchain, checking it corresponds to

the hash for the last digest in the list of digests obtained from the server, and running the verification in Section 6.5.1. The blockchain guarantees that all users see the same checkpoint. This prevents the server from forking two users' views, as the latest digests for two different views cannot both match the published checkpoint. In this way, we bootstrap the blockchain's consistency guarantees to achieve verifiable consistency over an entire epoch of operations.

6.5.3 Multiple Objects per Checkpoint

So far, the server puts one checkpoint in the blockchain *per object*, which is undesirable when there are many objects. We address this as follows. The server computes the hash of the final digest of each object, builds a Merkle tree over those hashes, and publishes the root hash in the blockchain as a single checkpoint for all objects. To verify integrity at the end of an epoch, a Ghostor client fetches the digest chain from the server for objects that are either (1) accessed by the client during the epoch or (2) owned by the client's user. It verifies that all operations that it performed on those objects are included in the objects' digest chains. Then, it requests Merkle proofs from the server to check that the hash of the latest digest is included in the Merkle tree at the correct position based on the object's PVK. Finally, it verifies that the Merkle root hash matches the published checkpoint.

Although we maintain a separate digest chain for each object, the collective history of operations, across all objects, is also linearizable. This follows from the classical result that linearizability is a local property [218]. Thus, Ghostor provides *verifiable linearizability across all objects, while supporting full concurrency for operations on different objects*.

6.5.4 Concurrent Operations on a Single Object

As explained in Section 6.5.1, the client must fetch the latest digest from the server to construct a digest for a new `GET` or `PUT`. If two clients attempt to `GET` or `PUT` an object concurrently, they may retrieve the same latest digest for that object, and therefore construct new digests that both have the same $\mathsf{Hash}_{\mathsf{prev}}$. An honest server can only accept one of them; the other operation must be aborted. A naïve fix is for clients to acquire locks (or leases) on objects during network round trips, but this limits single-object throughput according to client round-trip times. How can we allow concurrent operations on a single object without holding server-side locks during round trips? We explain our techniques at a high level below; Section 6.7 contains a full description of our protocol.

6.5.4.1 `GET`s

We optimize `GET`s so that clients need not fetch the latest digest, obviating the need to lock for a round trip. When a client submits a `GET` request to the server, the client need not include $\mathsf{Hash}_{\mathsf{prev}}$, $\mathsf{Hash}_{\mathsf{data}}$, or $\mathsf{Hash}_{\mathsf{keylist}}$ in the digest presented to the server. The client includes the remaining fields and a signature over only those fields. Then, the server chooses the hashes for the client and returns the resulting digest, signed by the server. Although the server can replay operations, this is harmless because `GET`s do not affect data. When the verification daemon verifies a `GET`, it checks the client signature without including $\mathsf{Hash}_{\mathsf{prev}}$, $\mathsf{Hash}_{\mathsf{data}}$, or $\mathsf{Hash}_{\mathsf{keylist}}$.

6.5.4.2 PUTs

The above technique does not apply to PUTs, because the server can roll back objects by replaying PUTs. Simply using a client-provided nonce to detect replayed PUTs is not sufficient, because the server can delay incorporating a PUT (which we call a *time-stretch* attack) to manipulate the final object contents. For PUTs, Ghostor uses a two-phase protocol. In the PREPARE phase, the client operates in the same way as GET, but signs the digest with WSK; the server fills in the hashes, signs the resulting digest, appends it to the object's digest chain, and returns it to the client. In the COMMIT phase, the client creates the final digest for the operation—omitting $\mathsf{Hash}_{\mathsf{prev}}$ and appending an additional field $\mathsf{Hash}_{\mathsf{prep}}$, which is the hash of the server-signed digest obtained in the PREPARE phase—and uploads it to the server with the new object contents. The server fills in $\mathsf{Hash}_{\mathsf{prev}}$ based on the object's digest chain (which could have changed since the PREPARE phase), signs the resulting digest, appends it to the object's digest chain, and returns it to the client. The server can replay PREPARE requests, but it does not affect object contents. The server cannot generate a COMMIT digest for a replayed PREPARE request, because the client signed the COMMIT digest including the hash of the server-signed PREPARE digest, which includes $\mathsf{Hash}_{\mathsf{prev}}$. The server can replay a COMMIT request for a particular PREPARE request, but this is harmless because of our conflict resolution strategy described below.

6.5.4.3 Resolving Conflicts

If two accesses are concurrent (i.e., neither commits before the other prepares), then linearizability does not require any particular ordering of those operations, only that all clients perceive the same ordering. If a GET is concurrent with a PUT (GET digest between the PREPARE and COMMIT digests for a PUT), Ghostor linearizes the GET as happening before the PUT. This allows the result of the GET to be served immediately, without waiting for the PUT to finish. For concurrent PUTs, it is unsafe for the linearization order to depend on the COMMIT digest, because the server could perform a time-stretch or replay attack on a COMMIT digest, to manipulate which PUT wins. Therefore, Ghostor chooses as the winning PUT the one whose PREPARE digest is latest. The server can still delay PREPARE digests, but the client can choose not to COMMIT if the delay is unacceptably large. To simplify the implementation of this conflict resolution procedure, we require that the PREPARE and COMMIT phases happen over the same session with the client, during which the server can keep in-memory state for the relevant object. This allows the server to match PREPARE and COMMIT digests without additional accesses to secondary storage.

6.5.4.4 Verification Complexity

To verify PUTs, the verification daemon must check that $\mathsf{Hash}_{\mathsf{data}}$ only changes on COMMIT digests for winning writes. Thus, it must keep track of all PREPARE digests since the latest PREPARE digest whose corresponding COMMIT has been seen. We can bound this state by requiring that PUT requests do not cross an epoch boundary.

6.5.4.5 ACL Updates

We envision that updates to the ACL will be rare, so our implementation does not allow **set_acl** operations to proceed concurrently with `GET`s or `PUT`s. It may be possible to apply a two-phase technique, similar to our concurrent `PUT` protocol, to allow **set_acl** operations to proceed concurrently with other operations. We leave exploring this to future work.

6.6 Mitigating Resource Abuse

To prevent resource abuse, commercial data-sharing systems, like Google Drive and Dropbox, enforce per-user resource quotas. Ghostor cannot do this, because Ghostor's anonymity prevents it from tracking users. Instead, Ghostor uses two techniques to prevent resource abuse without tracking users: anonymous payments and proof of work.

6.6.1 Anonymous Payments

A strawman approach is for users to use an anonymous cryptocurrency (e.g., Zcash [506]) to pay for each expensive operation (e.g., operations that consume storage). Unfortunately, this requires a separate blockchain transaction for each operation, limiting the system's overall throughput.

Instead, Ghostor lets users pay for expensive operations *in bulk* via the **pay** API call (Section 6.2). The server responds with a set of *tokens* proportional to the amount paid via Zcash, which can later be redeemed *without using the blockchain* to perform operations. Done naïvely, this violates Ghostor's anonymity; the server can track users by their tokens (tokens issued for a single **pay** call belong to the same user).

To circumvent this issue, Ghostor uses *blind signatures* [95, 105, 104]. A Ghostor client generates a random token and *blinds* it. After verifying that the client has made a cryptocurrency payment, the server signs the blinded token. The blind signature protocol allows the client to *unblind* it while preserving the signature. To redeem the token, the client gives the unblinded signed token to the server, who can verify the server's signature to be sure it is valid. The server cannot link tokens at the time of use to tokens at the time of issue because the tokens were blinded when the server originally signed them.

6.6.2 Proof of Work (PoW)

Another way to mitigate resource abuse is **proof of work (PoW)** [30]. Before each request from the client, the server sends a random $\mathsf{challenge}$ to the client, and the client must find a proof such that $\mathsf{Hash}(\mathsf{challenge}, \mathsf{proof}, \mathsf{request}) < \mathsf{diff}$. diff controls the difficulty, which is chosen to offset the amplification factor in the server's work. Because of the guarantees of the hash function, the client must iterate through different proofs until it finds one that works. In contrast, the server efficiently checks the proof by computing one hash.

6.6.3 Using Anonymous Payments and Proof of Work Together

Ghostor uses anonymous payments and PoW together to mitigate resource abuse. Our implementation requires anonymous payment only for **create_object**, which requires the server to commit additional storage space for the new object. This is analogous to systems like Google Drive or Dropbox, which require payment to increase a user's storage limit but do not charge based on the count or frequency of object accesses. Implicit in this model are hard limits on object size and per-object access frequency, which Ghostor can enforce. Although our implementation requires payment only for **create_object**, an alternate implementation may choose to require payment for every operation except **pay**. Ghostor requires PoW for all API calls. This includes **pay** and **create_object**, to offset the cost of Zcash payments and verifying blind signatures.

6.7 Full Protocol Description

Below, we describe the client-server protocol used by Ghostor.

6.7.1 `GET` Protocol

1. Server sends a PoW challenge to the client (Section 6.6).
2. Client sends the server the PoW solution, PVK of the object that the user wishes to access, and the server returns the object header and current epoch.
3. The client assembles a digest for the `GET` operation, including the epoch number, PVK, RVK, WVK, and a random nonce, and signs it with RSK (obtained from the header). It sends the signed digest to the server.
4. Server reads the latest digest and checks that the client's candidate digest is consistent with it. If not (for example, if the header was changed in-between round trips), the server gives the client the object header, and the protocol returns to Step 3.
5. Server adds $\mathsf{Hash}_{\mathsf{prev}}$, $\mathsf{Hash}_{\mathsf{header}}$, and $\mathsf{Hash}_{\mathsf{data}}$ to the digest (according to the order in which it commits operations on the object). Then it signs it and adds it to the log of digests for that object.
6. Server returns the object contents and the digest, including the server's signature, to the client.
7. Client checks that the signed digest matches the object contents and digest that the client provided. If so, it returns the object contents to the user and sends the signed digest to the verification daemon.

6.7.2 `PUT` Protocol

1. Server sends a PoW challenge to the client (Section 6.6).
2. Client sends the server the PoW solution and PVK of the object to `PUT`, and the server returns the object header, current epoch, and latest server-signed digest for that object.

3. The client assembles a PREPARE digest for the write operation, including the epoch number, PVK, RVK, WVK, and a random nonce, and signs it with WSK (obtained from the header). It sends the signed digest to the server.
4. Server reads the latest digest and checks that the client's candidate digest is consistent with it. If not, then the server gives the client the object header, and the protocol returns to Step 3.
5. Server adds $\mathsf{Hash}_{\mathsf{prev}}$, $\mathsf{Hash}_{\mathsf{header}}$, and $\mathsf{Hash}_{\mathsf{data}}$ to the digest (according to the order in which it commits operations on the object). Then it signs it and adds it to the log of digests for that object.
6. Server returns the signed digest to the client.
7. Client assembles a COMMIT digest for the write operation, including the same fields as the PREPARE digest, and also $\mathsf{Hash}_{\mathsf{prep}}$ and $\mathsf{Hash}_{\mathsf{data}}$ according to the new data. Then it signs it and uploads it to the server, including the new object contents.
8. Server decides if this PUT "wins." It wins as long as no other PUT whose PREPARE digest is after this PUT's PREPARE digest has already committed. If this PUT wins, then the server performs the write, signs the digest, and adds it to the log of digests for that object. If not, it still signs the digest and adds it to the log, but it replaces $\mathsf{Hash}_{\mathsf{data}}$ with the current hash of the data, including the value provided by the client as an "addendum" so that the verification daemon can still verify the client's signature. The server may also reject the COMMIT digest if the key list changed meanwhile due to a **set_acl** operation.
9. Server returns the digest, including the server's signature, to the client.
10. Client checks that the signed digest matches the object contents and digest that the client provided. If so, it sends the signed digest to the verification daemon.

6.7.3 Access Control

1. Server sends a PoW challenge to the client (Section 6.6).
2. Client sends the server the PoW solution and PVK of the object, and the server returns the object header, current epoch, and latest server-signed digest for that object.
3. Client samples fresh keys for the file (including RSK, WSK, and OSK, but not PSK), encrypts the object contents with OSK, and assembles a new header according to the new ACL, randomly shuffling the key list in the header and padding it to a maximum size if desired. The client assembles a digest for the operation, including all fields in Table 6.3, and signs it with PSK. It sends the signed digest to the server. The client also signs PVK with PSK and includes that signature in the request.
4. Server acquires a lock (lease) on the object for this client (unless it is already held for this client), reads the latest digest, and checks that the client's candidate digest is consistent with it. If not, then the server gives the client the object header, and the protocol returns to Step 3. When returning to Step 3, the server checks if the client's signature over PVK is correct. If so, the server holds the lock on the object during the round trip. If not, the server releases it.
5. Server updates the header and object contents, signs the digest, adds it to the log of digests for that object, and releases the lock.
6. Server returns the digest, including the server's signature, to the client.

7. Client checks that the signed digest matches the object contents and digest that the client provided. If so, it returns the object contents to the user and sends the signed digest to the verification daemon.

6.7.4 Object Creation

1. Server sends a PoW challenge to the client (Section 6.6)
2. Client sends the server the PoW, PVK of the object that the user wishes to create, a token signed by the server for proof of payment (Section 6.2), the header for the new object, and the object's first digest (for which $\mathsf{Hash}_{\mathsf{prev}}$ is empty). This involves generating all the keys in Figure 6.5) for the new object.
3. Server verifies the signature on the token, and checks that it has not been used before.
4. Server "remembers" the hash of the token by storing it in permanent storage.
5. Server writes the object header. It signs the digest and creates a log for this object containing only that digest.
6. Server returns the digest, including the server's signature, to the client.
7. Client checks that the signed digest matches the object contents and digest that the client provided. If so, it returns the object contents to the user and sends the signed digest to the verification daemon.

6.7.5 Verification Procedure

At the end of each epoch, the verification daemon downloads the digest chain and checkpoints to verify operations performed in the epoch.

1. Server sends a PoW challenge to the daemon (Section 6.6). (The server will request additional PoWs for long lists of digests as it streams them to the daemon in Step 3.)
2. Daemon responds with PoW and requests the object's digest chain from the server for that epoch. It sends the server a signed digest for that object, so the server knows this is a legitimate request.
3. Server returns the digest chain for that object, along with a Merkle proof.
4. Daemon retrieves the Merkle root from the checkpoint in Zcash, and verifies the server's Merkle proof to check that the last digest in the digest chain is included in the Merkle tree at the correct position based on the object's PVK.
5. Daemon verifies that all digests corresponding to the user's operations are in the digest chain, and that the digest chain is valid.

To check that the digest chain is valid, the daemon checks:

1. $\mathsf{Hash}_{\mathsf{prev}}$ for each digest matches the previous digest. If this digest is the first digest in this epoch, the previous digest is the last digest in the previous epoch. The daemon knows this previous digest already since the daemon must have checked the previous epoch. If this is the first epoch, then $\mathsf{Hash}_{\mathsf{prev}}$ should be empty.
2. $\mathsf{Hash}_{\mathsf{prep}}$ in each COMMIT digests matches an earlier PREPARE digest in the same epoch, and each PREPARE digest matches with at most one COMMIT digest.

3. $\mathsf{Hash}_{\mathsf{data}}$ only changes in winning COMMIT digests, which are signed with WSK.
4. WVK, RVK, and $\mathsf{Hash}_{\mathsf{keylist}}$ only change in digests signed with PSK, and PVK never changes.
5. The epoch number in digests matches the epoch that the client requested, and never decreases from one digest to the next.
6. $\mathsf{Sig}_{\mathsf{client}}$ is valid and signed using the correct signing key. For example, if this operation is read, $\mathsf{Sig}_{\mathsf{client}}$ must be signed using RSK.

6.7.6 Payment

First, the user pays the server using an anonymous cryptocurrency such as Zcash [506], and obtains a proof of payment from Zcash. Then, the client obtains tokens from the server, as follows:

1. Server sends a PoW challenge to the client (Section 6.6).
2. Client sends the server the PoW, proof of payment, and t blinded tokens, where t corresponds to the amount paid.
3. Server checks that the proof of payment is valid and has not been used before.
4. Server "remembers" the proof of payment by storing it in persistent storage.
5. Server signs the blinded tokens, ensuring that t indeed corresponds to the amount paid, and sends the signed blinded tokens to the client.
6. Client unblinds the signed tokens and saves them for later use.

6.8 Applying Ghostor to Applications

In this section, we discuss applying Ghostor to an EHR Sharing application. In the full Ghostor paper, we also discuss how to combine Ghostor's anonymity techniques with a globally oblivious scheme, AnonRAM [31], to obtain a *metadata-hiding* object-sharing scheme, *Ghostor-MH*. This dissertation does not include an in-depth discussion of Ghostor-MH because Ghostor-MH is a theoretical scheme, not a practical system.

Our goal in this section is to show how a real application may interface with Ghostor's semantics (e.g., ownership, key management, error handling) and how Ghostor's security guarantees might benefit a real application. To make the discussion concrete, we explore a particular use case: multi-institutional sharing of electronic health records (EHRs). It has been of increasing interest to put patients in control of their data as they move between different healthcare providers [202, 424, 223]. As it is paramount to protect medical data in the face of attackers [135], various proposals for multi-institutional EHR sharing use a blockchain for access control and integrity [340, 28]. Below, we explore how to design such a system using Ghostor to store EHRs in a central object store, using only decentralized trust. We also implemented the system for Open mHealth [359].

Each patient owns one or more objects in the central Ghostor system representing their EHRs. Each patient's Ghostor client (on her laptop or phone) is responsible for storing the PSKs for these objects. The PSKs could be stored in a wristband, as in [340], in case of emergency situations for at-risk patients. When the patient seeks treatment from a healthcare provider, she can grant the healthcare provider access to the objects containing the relevant information in Ghostor. Each

healthcare provider's Ghostor client maintains a local *metadata database*, mapping patient identities (object IDs, Section 6.2) to PVKs. This mapping could be created when a patient checks in to the office for the first time (e.g., by sharing a QR code).

Benefits. Existing proposals leverage a blockchain to achieve integrity guarantees [340, 28] but use the blockchain more heavily than Ghostor: for example, they require a blockchain transaction to grant access to a healthcare provider, which results in poor performance and scalability. Additionally, Ghostor provides anonymity for sharing records.

Epoch Time. An important aspect of Ghostor's semantics is that one has to wait until the next epoch before one can verify that no fork has occurred. It is reasonable to fetch a patient's record at the time that they check in to a healthcare facility, but before they are called in for treatment. This allows the time to wait until the end of an epoch to overlap with the patient's waiting time. In the case of scheduled appointments, the record can be fetched in advance so that integrity can be verified by the time of the appointment. An epoch time of 15–30 minutes would probably be sufficient. By tying the frequency of blockchain operations to a tunable parameter (the epoch time), Ghostor allows the cost of using expressive cryptography to be tuned, to strike the right balance between cost and verification delay for each application.

Error Handling. If a healthcare provider detects a fork when verifying an epoch, it informs other healthcare providers of the integrity violation out-of-band of the Ghostor system. Ghostor does not constrain what happens next. One approach, used in Certificate Transparency (CT), is to abandon the Ghostor server for which the integrity violation was detected. We envision that there would be a few Ghostor servers in the system, similar to logs in CT, so this would require affected users to migrate their data to a new server. Another approach is to handle the error in the same way that blockchain-based systems [340, 28] handle cases where the hash on the blockchain does not match the hash of the data—treat it as an availability error. While neither solution is ideal, it is better than the status quo, in which a malicious adversary is free to perform fork or rollback attacks undetected, causing patients to receive incorrect treatments based on old or incorrect data, potentially resulting in serious physical injury.

6.9 Implementation

We implemented a prototype of Ghostor in Go. It consists of three parts, as in Figure 6.4, server ($\approx$ 2100 LOC), client library ($\approx$ 1000 LOC), and verification daemon ($\approx$ 1000 LOC), which all depend on a set of core Ghostor libraries ($\approx$ 1400 LOC).

Our implementation uses Ceph RADOS [480] for consistent, distributed object storage. We use SHA-256 for the cryptographic hash and the NaCl secretbox library (which uses XSalsa20 and Poly1305) for authenticated symmetric-key encryption. For *key-private* asymmetric encryption (to encrypt signing keys in the object header), we implemented the ElGamal cryptosystem, which is *key-private* [44], on top of the Curve25519 elliptic curve. We use an existing blind signature implementation [401] based on RSA with 2048-bit keys and 1536-bit hashes. We use Ed25519 for digital signatures.

As discussed in Section 6.3, Ghostor uses external systems for anonymous communication and payment. In our implementation, clients use Tor [144] to communicate with the server and Zcash 1.0.15 for anonymous payments. We build a Zcash test network, separate from the Zcash main network. Ghostor, however, could also be deployed on the Zcash main chain. Zcash is also used as the blockchain to post checkpoints. Our implementation runs as a *single* Ghostor server that stores its data in a scalable, fault-tolerant, distributed storage cluster. We discuss how to scale to *multiple* servers in Section 6.11.2.

6.10 Evaluation

We run experiments on Amazon EC2. Except in Section 6.10.3, Ghostor's storage cluster consists of three `i3en.xlarge` servers. Additionally, we configure Ceph to replicate each object (key-value pair) on two SSDs on different machines, for fault-tolerance.

6.10.1 Microbenchmarks

Basic Crypto Primitives. We measured the latency of crypto operations used in Ghostor's critical path. En/decryption of object contents varies linearly with the object size, and takes ≈ 2 ms for 1 MiB. Key-private en/decryption for object headers and signing/verification of digests takes less than 150 us.

Blind Signatures. We also measure the blind signature scheme used for object creation, which consists of four steps. (1) The client *generates* a blinded hash of a random number. (2) The server *signs* the blinded hash. (3) The client *unblinds* the signature, obtaining the server's signature over the original number. (4) The server *verifies* the signature and the number during object creation. Results are shown in Figure 6.6.

Verification Procedure. In Figure 6.8, we measure the overhead of verification for digests in a single epoch. For client verification time, we perform an end-to-end test, measuring the total time to fetch digests and to verify them. The client has 1,000 signed digests for operations the client performed during the epoch that the client needs to check were included in the history of digests. We vary the total number of digests in the object's history for that epoch. The reported values in Figure 6.8a are the total time to verify the object, divided by the total number of operations on the object, indicating the verification time *per digest*. The trend indicates a constant overhead when the total number of operations on the object is small, that is amortized when the number of operations is large.

Figure 6.8b shows the server's overhead to compute the Merkle root. We inserted objects using YCSB (Section 6.10.2.2) during an epoch, and measured the time to compute the Merkle root at the end of that epoch. For 10,000 objects, this takes about 2.5 seconds; for 1,000,000 objects, it takes about 280 seconds. Reading the latest digest for each object (leaves of the Merkle tree) dominates the time to compute the Merkle root (2 seconds for 10,000 objects, 272 seconds for 1,000,000 objects). The reason is that our on-disk data structures are optimized for single-object

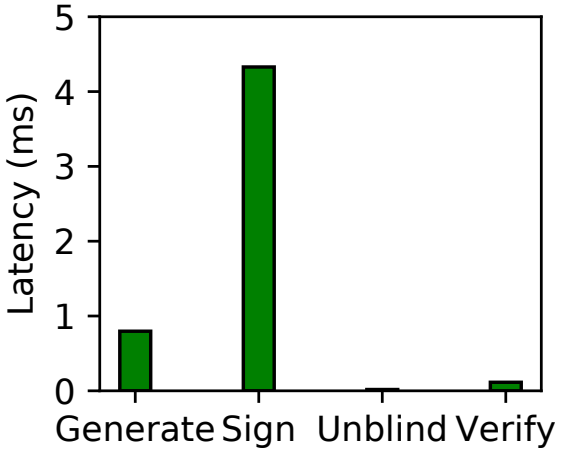
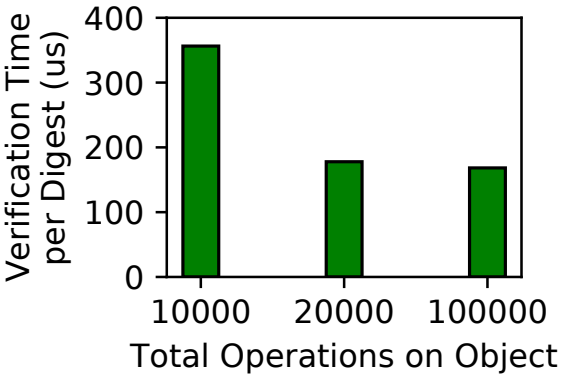


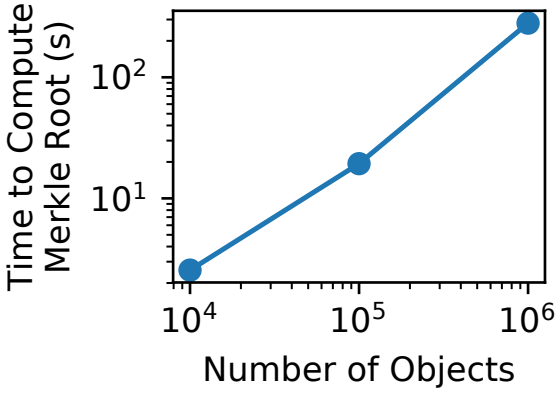
Figure 6.6: Blind signature.

A	50% R, 50% W
B	95% R, 5% W
C	100% R
D	95% R, 5% Insert
E	95% R, 5% Range
F	50% R, 50% R-Modify-W

Figure 6.7: YCSB workloads (R: read, W: write).



(a) Run verification procedure.



(b) Compute Merkle root.

Figure 6.8: Operations for verification.

operations, which are in the critical path. In particular, each object's digest chain is stored as a separate batched linked list, so reading the latest digests requires a separate read for each object.

6.10.2 Server-Side Overhead

This section measures to what extent anonymity and VerLinear affect Ghostor's performance. To ensure that the bottleneck was on the server, we set proof of work to minimum difficulty and do not use anonymous communication (Section 6.3), but we return to evaluating these in Section 6.10.3.

We measure the end-to-end performance of operations in Ghostor, both as a whole and for

instantiations of Ghostor having only anonymity or VerLinear. We compare these to an insecure baseline as well as to competitive solutions for privacy and verifiable consistency, as we now describe.

1. Insecure system ("Insec"). This system uses the traditional ACL-based approach for serving objects. Each object access is preceded by a read to the object's ACL to verify that the user has permission to access the object. Similarly, creating an object requires a read to a per-user account file. It provides no security against a compromised server.

2. End-to-End Encrypted system ("E2EE"). This system encrypts objects placed on the server using end-to-end encryption similarly to SiRiUS [190]. Such systems have an encrypted KeyList similar to Ghostor's, but clients can cache their keys locally on most accesses unlike Ghostor.

3. Ghostor's anonymity system ("Anon"). This is Ghostor with VerLinear disabled. This fits a scenario where one wants to hide information from a *passive* server attacker. Unlike the E2EE system above, this system cannot cache keys locally—every operation incurs an additional round trip to fetch the KeyList from the server. In addition, every operation incurs yet another round trip at the beginning for the client to perform a proof of work. On the positive side, the server does not maintain any per-user ACL.

4. Fork Consistent system ("ForkC"). This system maintains Ghostor's digest chain (Section 6.5.1), but does not post checkpoints. Each operation appends to a per-object log of digests, using the techniques in Section 6.5.4. This system also performs an ACL check when creating an object.

5. Ghostor's VerLinear system ("VLinear"). This system corresponds to the VerLinear mechanism in Section 6.5 (including Section 6.5.2). This matches a use case where one wants integrity, but does not care about privacy. We do not include the verification procedure, already evaluated in Section 6.10.1.

6. Ghostor. This system achieves both anonymity and VerLinear, and therefore incurs the costs of both guarantees.

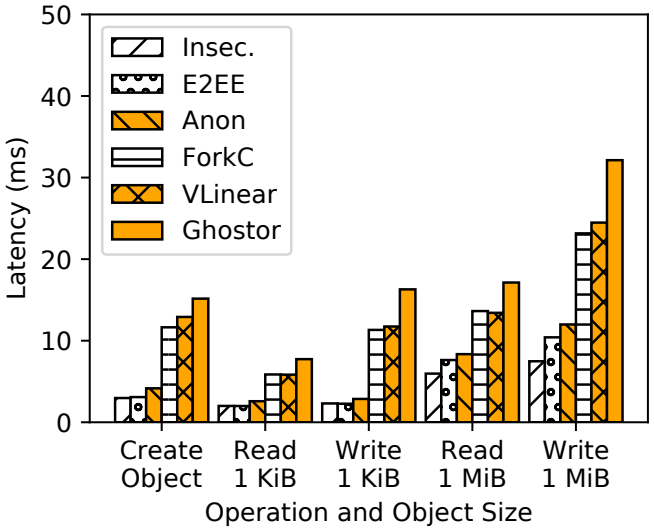
6.10.2.1 Object Accesses

In each setup, we measured the latency for create, `GET`, and `PUT` operations (Figure 6.9a), throughput for `GET`s/`PUT`s to a single object (Figure 6.10a), and the throughput for creating objects and for `GET`s/`PUT`s to multiple objects (Figure 6.10b).

Fork consistency adds substantial overhead, because additional accesses to persistent storage are required for each operation, to maintain each object's log of digests. Ghostor, which both maintains a per-object log of digests and provides anonymity, incurs additional overhead because clients do not cache keys, requiring the server to fetch the header for each operation. In contrast, for Anon, the additional cost of reading the header is offset by the lack of ACL check. For 1 MiB objects, en/decryption adds a visible overhead to latency.

End-to-end encryption adds little overhead to throughput; this is because we are measuring throughput at the *server*, whereas encryption and decryption are performed by *clients*. The only factor affecting server performance is that the ciphertexts are 40 bytes larger than plaintexts.

Single-object throughput is lower for ForkC, VLinear, and Ghostor, because maintaining a digest chain requires requests to be serialized across multiple accesses to persistent storage. In



(a) Latency benchmarks.

Operation	Latency (ms)
Proof of Work	0.57
Read Header	1.1
Client Processing	0.68
Check Client Digest	0.14
Read/Fill Digest	3.2
Append Digest	1.5
Read Data	2.1
Client Processing	9.1

(b) Latency Breakdown for Ghostor, Read 1 MiB.

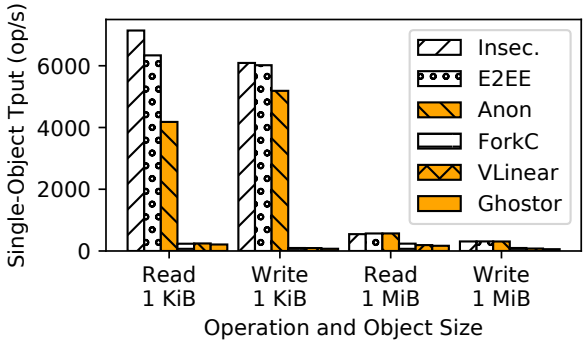
Figure 6.9: Latency measurements.

contrast, Insec, E2EE, and Anon serve requests in parallel, relying on Ceph's internal concurrency control.

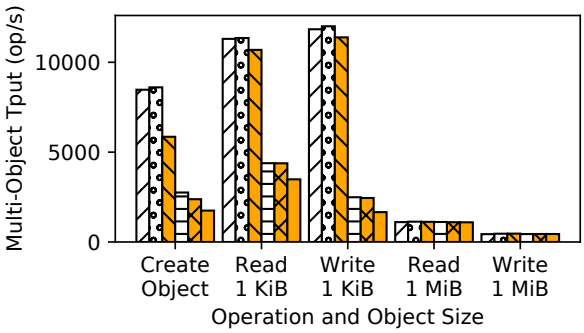
In the multi-object experiments, in which no two concurrent requests operate on the same object, this bottleneck disappears. For small objects, throughput drops in approximately an inverse pattern to the latency, as expected. For large objects, however, all systems perform commensurately. This is likely because reading/writing the object itself dominated the throughput usage for these experiments, without any concurrency overhead at the object level to differentiate the setups.

6.10.2.2 Yahoo! Cloud Serving Benchmark

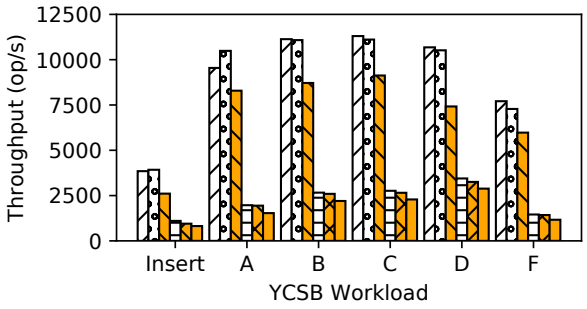
In this section, we evaluate our system using the Yahoo! Cloud Serving Benchmark (YCSB). YCSB provides different workloads representative of various use cases, summarized in Table 6.7. We do not use Workload E because it involves range queries, which Ghostor does not support. As shown in Figure 6.10c, anonymity incurs up to a 25% overhead for benchmarks containing insertions, owing to the additional accesses to storage required to store used object creation tokens. However, it shows essentially no overhead for `GET`s and `PUT`s. Fork consistency adds a 3–4× overhead compared to the Insec baseline. VerLinear adds essentially no overhead on top of fork consistency; this is to be expected, because the overhead of VerLinear is outside of the critical path (except for insertions, where the overhead is easily amortized). Ghostor, which provides both anonymity and VerLinear, must forgo client-side caching, and therefore incurs additional overhead, with a 4–5× throughput reduction overall compared to the Insec baseline.



(a) Single-Object Throughput.



(b) Multi-Object Throughput.



(c) Throughput for YCSB.

Figure 6.10: Benchmarks comparing throughput of the six setups described in Section 6.10.2.

6.10.3 End-to-End Latency

We analyze Ghostor's performance from the client's perspective, including PoW and anonymous communication (Section 6.3). In these experiments, we use three `m4.10xlarge` instances each with three gp2 SSDs for Ghostor's storage cluster.

6.10.3.1 Microbenchmarks

The latency experienced by a Ghostor client is the latency measured in Figure 6.9, plus the additional overhead due to the proof of work mechanism and anonymous communication. The difficulty of the proof of work problem is adjustable. For the purpose of evaluation, we set it to a realistic value to prevent denial of service. Figure 6.9b indicates that it takes ≈ 32 ms for a Ghostor operation; therefore, we set the proof of work difficulty such that it takes the client, on average, 100 times longer to solve (≈ 3.2 s). Figure 6.11 shows the distribution of latency for the client to solve the proof of work problem. As expected, the distribution appears to be memoryless.

In our implementation, a client connects to a Ghostor server by establishing a circuit through the Tor [144] network. The performance of the connection, in terms of both latency and throughput, varies according to the circuit used. Figure 6.11 shows the distribution of (1) circuit establishment

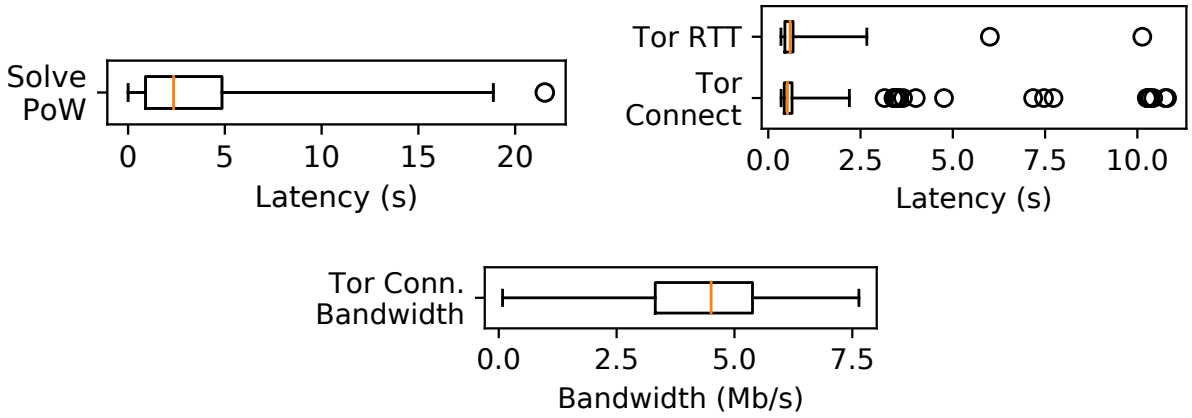


Figure 6.11: Microbenchmarks of PoW mechanism and Tor.

time, (2) round-trip time, and (3) network bandwidth. We used a fresh Tor circuit for each measurement. Based on our measurements, a Tor circuit usually provides a round-trip time less than 1 second and bandwidth of at least 2 Mb/s.

6.10.3.2 Macrobenchmarks

We now measure the end-to-end latency of each operation in Ghostor's client API (Section 6.2), including all overheads experienced by the client. As explained in Section 6.10.3.1, the overhead due to proof of work and Tor is quite variable; therefore, we repeat each experiment 1000 times, using a separate Tor circuit each time, and report the distribution of latencies for each operation in Figure 6.12. Comparing Figure 6.12 to Figure 6.9, the client-side latency is dominated by the cost of PoW and Tor; Ghostor's core techniques in Figure 6.9 have relatively small latency overhead. For the `pay` operation, we measure only the time to redeem a Zcash payment for a single token, not the time for proof of work or making the Zcash payment (see Section 6.10.4 for a discussion of this overhead). `GET` and `PUT` for large objects are the slowest, because Tor network bandwidth becomes a bottleneck. The `create_user` operation (not shown in Figure 6.12) is only 132 microseconds, because it generates an ElGamal keypair locally without any interaction with the server.

6.10.4 Zcash

In our implementation, we build our own Zcash test network to avoid the expense from Zcash's main network. Since our system leverages Zcash in a minimal way, the overhead of Zcash is not on the critical path of our protocol. According to the Zcash website [506] and block explorer [55], the block size limit is about 2 MiB, and block interval is about 2.5 minutes. In the past six months, the maximum block size has been less than 150 KiB and the average transaction fee has been much less than 0.001 ZEC (0.05 USD at the time of writing). Hence, even with shorter epochs (less time

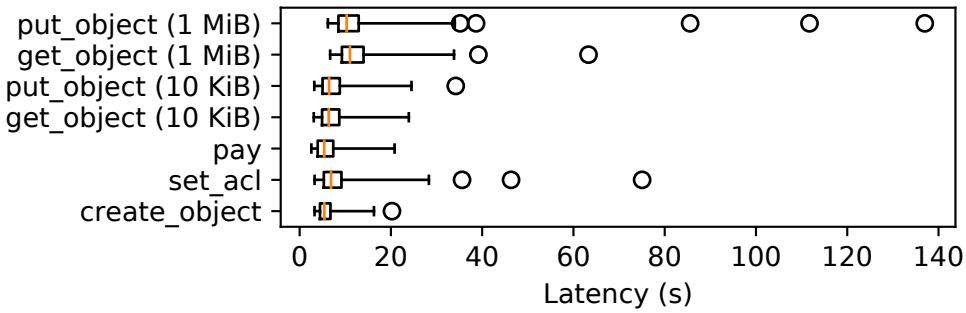


Figure 6.12: End-to-end latencies of client-side operations.

for misbehavior detection), the price of Ghostor's checkpoints is modest since there is a single checkpoint per epoch for the whole system.

6.11 Extensions

We briefly discuss two possible extensions to our Ghostor design. The first has to do with supporting a hierarchical directory structure, and the second has to do with scaling Ghostor to multiple servers.

6.11.1 Files and Directories

Our design of Ghostor can be extended to support a hierarchy of directories and files. Each directory or file corresponds to a PVK and associated Ghostor object; the PVK has a similar role to an inode number in a traditional file system. The Ghostor object corresponding to a directory contains a mapping from name to PVK as a list of *directory entries*. Given the PVK of a root directory and a filepath, a client iteratively finds the PVK of each directory from left to right; in the end, it will have the PVK of the file, allowing it to access the Ghostor object corresponding to a file. The procedure is analogous to resolving a filepath to an inode number in a traditional file system. The Ghostor object corresponding to a file may either contain the file contents directly, or it may contain the PVKs of other objects containing the file data, like an inode in a traditional file system.

The "no user-side caching" principle in Section 6.4 applies here, in the sense that clients may not cache the PVK of a file after resolving it once. A client must re-resolve a file's PVK on each access; caching the PVK and accessing the file without first accessing all parent directories would reveal that the same user has accessed the file before.

6.11.2 Scalability

Our implementation of Ghostor that we evaluated in Section 6.10 consists of a single Ghostor server, which stores data in a storage cluster that is internally replicated and fault-tolerant (Ceph RADOS). In this section, we discuss techniques to scale this setup by replicating the Ghostor server as well.

Given that we consider a malicious adversary, it may seem natural to use PBFT [101]. PBFT, however, is neither necessary nor sufficient in Ghostor's setting. It is not necessary because we already post checkpoints to a ledger based on decentralized trust (Section 6.5.2) to achieve verifiable integrity. It is not sufficient because we assume an adversary who can compromise any few machines across which we replicate Ghostor, which is incompatible with Byzantine Fault Tolerance.

The primary challenge to replicating the Ghostor server is *synchronization*: if multiple operations on the same object may be handled by different servers, the servers may concurrently mutate the on-disk data structure for that object. A simple solution is to use object-level locks provided by Ceph RADOS. This is probably sufficient for most uses. But, if server-side caching of objects in memory is implemented, caches in the Ghostor servers would have to be kept coherent.

Alternatively, one could partition the object space among the servers, so each object has a single server responsible for processing operations on it. A set of *load balancer* servers run Paxos [303, 302] to arrive at a consensus on which servers are up and running, so that requests meant for one server can be re-routed to another if it goes down. Note that Paxos is outside of the critical path; it only reacts to failures, not to individual operations. Based on the consensus, the load balancers determine which server is responsible for each object. Because all objects are stored in the same storage pool, the objects themselves do not need to be moved when Ghostor servers are added or removed, only when storage servers are added or removed (which is handled by Ceph). Object-level locks in Ceph RADOS would still be useful to enforce that at most one server is operating on a Ghostor object at a time.

6.12 Related Work

Existing work has looked at providing integrity in the presence of a malicious server. We have already compared extensively with SUNDR [315]. Venus [429] achieves eventual consistency; however, Venus requires some clients to be frequently online and is vulnerable to malicious clients. Caelus [273] has a similar requirement and does not resist collusion of malicious clients and the server. Verena [262] trusts one of two servers. SPORC [167], which combines fork consistency with operational transformation, allows clients to recover from a fork attack, but does not resist faulty clients. Depot [329] can tolerate faulty clients, but achieves a weaker notion of consistency than VerLinear. Furthermore, its consistency techniques are at odds with anonymity. Ghostor and these systems use hash chains [208, 334] as a key building block.

Many systems provide end-to-end encryption (E2EE), but leak significant user information as discussed in Section 6.3.3. These include academic systems such as Persona [32], DEPSKY [50], CFS [57], SiRiUS [190], Plutus [261], ShadowCrypt [216], M-Aegis [304], Mylar [382] and

Sieve [468] and industrial systems such as Crypho [131], Tresorit [453], Keybase [269], PreVeil [383], Privly [384] and Virtru [461].

Some systems, such as Haven [39] and A-SKY [124], protect against a malicious server by using trusted hardware. Such systems require trust in the hardware provider (e.g., Intel, if using Intel SGX). Furthermore, existing trusted hardware, like Intel SGX, is susceptible to side-channel attacks [92].

A complementary line of work to Ghostor aims to hide access patterns: *which* object was accessed. Standard Oblivious RAM (ORAM) [193, 427, 476] works in the single-client setting. Multi-client ORAM [31, 211, 263, 327, 328, 405, 440] extends ORAM to support multiple clients. These works either rely on central trust [405, 440] (either a fully trusted proxy or fully trusted clients) or provide limited functionality (not providing global object *sharing* [31], or revealing user identities [327]). GORAM [328] assumes the adversary controlling the server does not collude with clients. Furthermore, it only provides obliviousness within a single data owner's objects, not *global obliviousness* across all data owners.

AnonRAM [31] and PANDA [211] provide global obliviousness and hide user identity, but they are slow. They do not provide for sharing objects or mitigating resource abuse. One can realize these features by applying Ghostor's techniques to these schemes, as we do in our full Ghostor paper [227] to build Ghostor-MH. Unlike these schemes, Ghostor-MH is a *metadata-hiding object-sharing scheme* providing both global obliviousness and anonymity without trusted parties or non-collusion assumptions.

Peer-to-peer storage systems, like OceanStore [290], Pastry [400], CAN [394], and IPFS [48], allow users to store objects on globally distributed, untrusted storage without any coordinating central trusted party. These systems are vulnerable to rollback/fork attacks on mutable data by malicious storage nodes (unlike Ghostor's VerLinear). While some of them encrypt objects for privacy, they do not provide a mechanism to distribute secret keys while preserving anonymity, as Ghostor does. Recent blockchain-based decentralized storage systems, like Storj [441], Swarm [454], Filecoin [171], and Sia [430], have similar shortcomings.

As discussed in Section 6.1, blockchain-based and BFT-based systems [101, 500, 353, 93] and verifiable ledgers [305, 342] can serve as the source of decentralized trust in Ghostor. Another line of work aims to provide efficient auditing mechanisms. EthIKS [73] leverages smart contracts [93] to monitor key transparency systems [342]. Catena [451] builds log systems based on Bitcoin transactions, which enables efficient auditing by low-power clients. It may be possible to apply techniques from those works to optimize our verification procedure in Section 6.5.2. However, none of them aim to build secure data-sharing systems like Ghostor.

Secure messaging systems [128, 455, 222] hide network traffic patterns, but they do not support object storage/sharing as in our setting. Ghostor can complementarily use them for its anonymous communication network.

6.13 Conclusion

Ghostor is a data-sharing system that provides *anonymity* and *verifiable linearizability* in a strong threat model that assumes only *decentralized trust*. Ghostor's decision to provide anonymity as its privacy guarantee is similar in spirit to the technique in Section 3.2.3, as it allows Ghostor to delink user identities from accessed data without using expensive cryptographic tools like multi-client ORAM. Although Ghostor leverages a blockchain in order to achieve its integrity guarantee, we apply techniques from Section 3.2.1 and Section 3.2.2 to use the blockchain in an efficient way. These techniques allow Ghostor to provide strong privacy and integrity guarantees with practical overhead.

Chapter 7

Using Cryptography Efficiently for Many-to-Many End-to-End Encryption for IoT

This is the second of two chapters exploring the techniques in Section 3.2. We focus in this chapter on publish-subscribe communication for industrial Internet of Things (IoT) deployments. Given that IoT deployments collect physical information, which is often privacy-sensitive, it is a natural goal to encrypt data end-to-end during transit. As we will see, policy-based encryption like Attribute-Based Encryption (ABE) (Section 2.1.1.2) is a natural fit for such deployments, as it would allow an end-to-end encrypted publish-subscribe system to support similar semantics to unencrypted IoT publish-subscribe systems. Unfortunately, ABE is too expensive, particularly for ultra low-power embedded devices with limited energy budgets.

In this chapter, we present the design, implementation, and evaluation of JEDI, an end-to-end encryption protocol for such IoT deployments. In designing JEDI, we apply all four techniques from Section 3.2 to reduce the cost of cryptography and arrive at a solution that supports similar semantics to unencrypted systems while being cheap enough to be practical even on ultra low-power deeply embedded sensor devices. We apply JEDI to an existing IoT messaging system, bw2, and demonstrate that its overhead is modest.

7.1 Introduction

As the Internet of Things (IoT) has emerged over the past decade, smart devices have become increasingly common. This trend is only expected to continue, with tens of billions of new IoT devices deployed over the next few years [118]. The IoT vision requires these devices to communicate to discover and use the resources and data provided by one another. Yet, these devices collect privacy-sensitive information about users. A natural step to secure privacy-sensitive data is to use *end-to-end encryption* to protect it during transit.

Existing protocols for end-to-end encryption, such as SSL/TLS and TextSecure [179], focus

on *one-to-one* communication between two principals: for example, Alice sends a message to Bob over an insecure channel. Such protocols, however, appear not to be a good fit for large-scale industrial IoT systems. These IoT systems demand *many-to-many* communication among **decoupled** senders and receivers, and require **decentralized delegation of access** to enforce which devices can communicate with which others. This chapter presents JEDI, an end-to-end encryption protocol for such IoT systems.

7.1.1 Requirements for JEDI

We investigate existing IoT systems, which currently do not encrypt data end-to-end, to understand the requirements on an end-to-end encryption protocol like JEDI. We use *smart cities* as an example application area, and data-collecting sensors in a large organization as a concrete use case. We identify three central requirements, which we treat in turn below.

7.1.1.1 Decoupled Senders and Receivers

IoT-scale systems could consist of thousands of principals, making it infeasible for consumers of data (e.g., applications) to maintain a separate session with each producer of data (e.g., sensors). Instead, senders are typically **decoupled** from receivers. Such decoupling is common in *publish-subscribe* systems for IoT, such as MQTT, AMQP, XMPP, and Solace [434]. In particular, many-to-many communication based on publish-subscribe is the *de-facto* standard in smart buildings, used in systems like BOSS [137], VOLTTRON [463], Brume [341] and bw2 [16], and adopted commercially in AllJoyn and IoTivity. Senders publish messages by addressing them to *resources* and sending them to a *router*. Recipients *subscribe* to a resource by asking the router to send them messages addressed to that resource.

Many systems for smart buildings/cities, like sMAP [136], SensorAct [19], bw2 [16], VOLTTRON [463], and BAS [289], organize resources as a **hierarchy**. A resource hierarchy matches the organization of IoT devices: for instance, smart cities contain buildings, which contain floors, which contain rooms, which contain sensors, which produce streams of readings. We represent each resource—a leaf in the hierarchy—as a Uniform Resource Indicator (**URI**), which is like a file path. For example, a sensor that measures temperature and humidity might send its readings to the two URIs `buildingA/floor2/roomLHall/sensor0/temp` and `buildingA/floor2/roomLHall/sensor0/hum`. A user can subscribe to a URI prefix, such as `buildingA/floor2/roomLHall/*`, which represents a subtree of the hierarchy. He would then receive all sensor readings in room "LHall."

[1]Image credits: `https://tweakers.net/pricewatch/1275475/asus-f540la-dm1201t.html`, `https://www.lg.com/uk/mobile-phones/lg-H791`, `https://www.bestbuy.com/site/nest-learning-thermostat-3rd-generation-stainless-steel/4346501.p?skuId=4346501`, `https://www.macys.com/shop/product/fitbit-charge-2-heart-rate-fitness-wristband?ID=2999458`

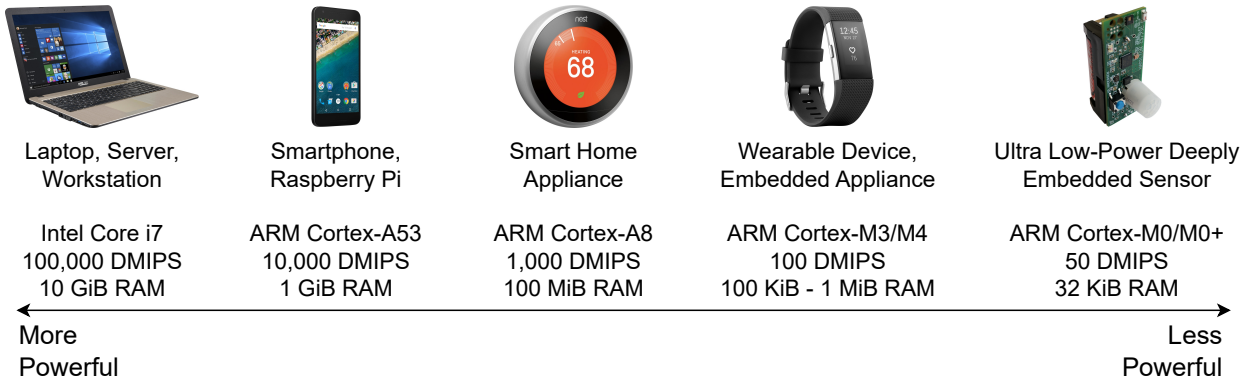


Figure 7.1: IoT comprises diverse devices that span more than four orders of magnitude of computing power (estimated in Dhrystone MIPS).[1]

7.1.1.2 Decentralized Delegation

Access control in IoT needs to be fine-grained. For example, if Bob has an app that needs access to temperature readings from a single sensor, that app should receive the decryption key for only that one URI, even if Bob has keys for the entire room. In an IoT-scale system, it is not scalable for a central authority to individually give fine-grained decryption keys to each person's devices. Moreover, as we discuss in Section 7.2, such an approach would pose increased security and privacy risks. Instead, Bob, who himself has access to readings for the entire room, should be able to delegate temperature-readings access to the app. Generally, a principal with access to a set of resources can give another principal access to a subset of those resources.

Vanadium [444] and bw2 [16] introduced *decentralized delegation* (SPKI/SDSI [121] and Macaroons [54]) in the smart buildings space. Since then, decentralized delegation has become the state-of-the-art for access control in smart buildings, especially those geared toward large-scale commercial buildings or organizations [170, 236]. In these systems, a principal can access a resource if there exists a *chain* of delegations, from the owner of the resource to that principal, granting access. At each link in the chain, the extent of access may be qualified by *caveats*, which add restrictions to which resources can be accessed and when. While these systems provide delegation of permissions, they do not provide protocols for encrypting and decrypting messages end-to-end.

7.1.1.3 Resource Constraints

IoT devices vary greatly in their capabilities, as shown in Figure 7.1. Some IoT devices, such as wearable devices and low-cost sensor platforms, are constrained in CPU, memory, and energy. In particular, ultra low-power deeply embedded sensing devices, at the far right of Figure 7.1, operate within the extreme resource constraints that we considered in the context of *TCPlp* in Chapter 5 (Section 5.2.1). An application of interest involving such devices is *indoor environmental sensing*

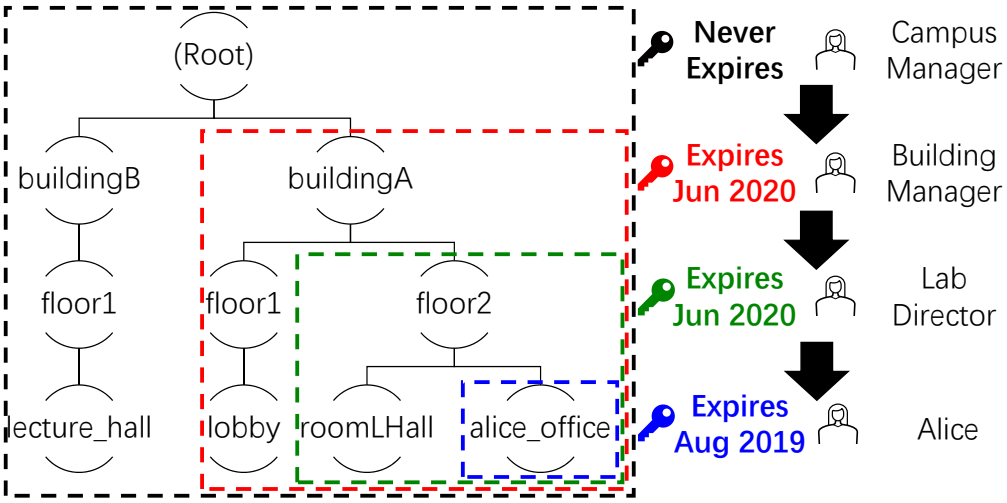


Figure 7.2: JEDI keys can be qualified and delegated, supporting decentralized, cryptographically-enforced access control via key delegation. Each person has a decryption key for the indicated resource subtree that is valid until the indicated expiry time. Black arrows denote delegation.

in smart buildings/cities. Sensors that measure temperature, humidity, or occupancy may be deployed in a building; such sensors are *battery-powered* and transmit readings using a *low-power* wireless network (i.e., LLNs). To see ubiquitous deployment, they must cost only *tens of dollars* per unit and have *several years* of battery life. To achieve this price/power point, sensor platforms are heavily resource-constrained, with mere *kilobytes* of memory (farthest right in Figure 7.1) [210, 370, 169, 314, 88, 15, 14]. The *power consumption* of encryption is a serious challenge, even more so than its latency on a slower CPU; the CPU and radio must be used sparingly to avoid consuming energy too quickly [498, 274]. For example, on the sensor platform used in our evaluation, an average CPU utilization of merely 5% would result in less than a year of battery life, *even if the power cost of using the transducers and network were zero*.

7.1.2 Overview of JEDI

This chapter presents JEDI, a *many-to-many* end-to-end encryption protocol compatible with the above three requirements of IoT systems. JEDI encrypts messages end-to-end for confidentiality, signs them for integrity while preserving anonymity, and supports delegation with caveats, all while allowing senders and receivers to be decoupled via a resource hierarchy. JEDI differs from existing encryption protocols like SSL/TLS, requiring us to overcome a number of *challenges*:

1. Formulating a new system model for end-to-end encryption to support **decoupled senders and receivers** and **decentralized delegation** typical of IoT systems (Section 7.1.2.1)

2. Realizing this expressive model while working within the **resource constraints** of IoT devices (Section 7.1.2.2)

3. Allowing receivers to verify the integrity of messages, while preserving the anonymity of senders (Section 7.1.2.3)

4. Extending JEDI's model to support revocation (not described in this dissertation; see our extended paper [297])

Below, we explain how we address each of these challenges.

7.1.2.1 JEDI's System Model (Section 7.2)

Participants in JEDI are called *principals*. Any principal can create a **resource hierarchy** to represent some resources that it owns. Because that principal owns all of the resources in the hierarchy, it is called the *authority* of that hierarchy.

Due to the setting of **decoupled senders and receivers**, the sender can no longer encrypt messages with the receiver's public key, as in traditional end-to-end encryption. Instead, JEDI models principals as interacting with resources, rather than with other principals. Herein lies the key difference between JEDI's model and other end-to-end encryption protocols: the publisher of a message encrypts it according to the URI to which it is published, not the recipients subscribed to that URI. Only principals permitted to subscribe to a URI are given keys that can decrypt messages published to that URI.

IoT systems that support **decentralized delegation** (Vanadium, bw2), as well as related non-IoT authorization systems (e.g., SPKI/SDSI [121] and Macaroons [54]) provide principals with tokens (e.g., certificate chains) that they can present to prove they have access to a certain resource. Providing tokens, however, is not enough for end-to-end encryption; unlike these systems, JEDI allows *decryption keys* to be distributed via chains of delegations. Furthermore, the URI prefix and expiry time associated with each JEDI key can be restricted at each delegation. For example, as shown in Figure 7.2, suppose Alice, who works in a research lab, needs access to sensor readings in her office. In the past, the campus facilities manager, who is the authority for the hierarchy, granted a key for `buildingA/*` to the building manager, who granted a key for `buildingA/floor2/*` to the lab director. Now, Alice can obtain the key for `buildingA/floor2/alice_office/*` directly from her local authority (the lab director).

7.1.2.2 Encryption with URIs and Expiry (Section 7.3)

JEDI supports *decoupled* communication. The resource to which a message is published acts as a *rendezvous point* between the senders and receivers, used by the underlying system to route messages. Central to JEDI is the challenge of finding an analogous *cryptographic rendezvous point* that senders can use to encrypt messages without knowledge of receivers. A number of IoT systems [418, 375] use only simple cryptography like AES, SHA2, and ECDSA, but these primitives are not expressive enough to encode JEDI's rendezvous point, which must support hierarchically-structured resources, non-interactive expiry, and decentralized delegation.

Existing systems [468, 470, 471] with similar expressivity to JEDI use Attribute-Based Encryption (ABE) [203, 51]. Unfortunately, ABE is not suitable for JEDI because it is too expensive,

especially in the context of **resource constraints** of IoT devices. Some IoT systems rule it out due to its latency alone [418]. In the context of low-power devices, encryption with ABE would also consume too much power. JEDI circumvents the problem of using ABE or basic cryptography with two insights: (1) Even though ABE is too heavy for low-power devices, this does not mean that we must resort to only symmetric-key techniques. We show that certain IBE schemes [2] can be made practical for such devices. (2) **Time is another resource hierarchy**: a timestamp can be expressed as `year/month/day/hour`, and in this hierarchical representation, any time range can be represented efficiently as a logarithmic number of subtrees. With this insight, we can simultaneously support URIs and expiry via a nonstandard use of a certain type of IBE scheme: WKD-IBE [2]. Like ABE, WKD-IBE is based on bilinear groups (pairings), but it is an order-of-magnitude less expensive than ABE as used in JEDI. This is an application of the technique from Section 3.2.3—we are leveraging the expressivity-efficiency trade-off (Section 2.2.2) to choose a cryptographic scheme that is cheaper but less expressive than ABE, yet expressive enough to provide similar semantics to IoT systems that do not encrypt data. To make JEDI practical on low-power devices, we design it to invoke WKD-IBE *rarely*, while relying on AES most of the time, much like session keys—an application of the technique from Section 3.2.1. The session keys must change, requiring another invocation of WKD-IBE, whenever the hierarchical representation of the timestamp changes—an application of the technique from Section 3.2.2, since it allows the frequency of WKD-IBE operations to be adjusted based on the granularity with which the timestamp is encoded. Additionally, we apply the technique from Section 3.2.4 Thus, JEDI achieves expressivity commensurate to IoT systems that do not encrypt data—significantly more expressive than AES-only solutions—while allowing several years of battery life for low-power low-cost IoT devices.

7.1.2.3 Integrity and Anonymity (Section 7.4)

In addition to being encrypted, messages should be signed so that the recipient of a message can be sure it was not sent by an attacker. This can be achieved via a certificate chain, as in SPKI/SDSI or bw2. Certificates can be distributed in a decentralized manner, just like encryption keys in Figure 7.2.

Certificate chains, however, are insufficient if anonymity is required. For example, consider an office space with an occupancy sensor in each office, each publishing to the same URI `buildingA/occupancy`. In aggregate, the occupancy sensors could be useful to inform, e.g., heating/cooling in the building, but individually, the readings for each room could be considered privacy-sensitive. The occupancy sensors in different rooms could use different certificate chains, if they were authorized/installed by different people. This could be used to deanonymize occupancy readings. To address this challenge, we adapt the WKD-IBE scheme that we use for end-to-end encryption to achieve an *anonymous* signature scheme that can encode the URI and expiry and support decentralized delegation. We apply the four techniques from Section 3.2 to signatures in analogous ways as we do for encryption, starting with using WKD-IBE, rather than a more costly cryptographic scheme, for anonymous signatures. As a result, anonymous signatures in JEDI are practical even on low-power embedded IoT devices.

7.1.2.4 Revocation

As stated above, JEDI keys support expiry. Therefore, it is possible to achieve a lightweight revocation scheme by delegating each key with short expiry and periodically renewing it to extend the expiry. To revoke a key, one simply does not renew it. We expect this expiry-based revocation to be sufficient for most use cases, especially for low-power devices that "sense and send."

In our extended paper [297], we provide a protocol for revoking keys *immediately*, without relying on expiry. As we discuss in Section 7.3.9, any cryptographically-enforced scheme that provides immediate revocation (i.e., keys can be revoked without waiting for them to expire) is subject to the fundamental limitation that the sender of a message must know which recipients are revoked when it encrypts the message. In our extended paper [297], we provide a protocol for immediate revocation in JEDI, subject to this constraint. We use techniques from tree-based broadcast encryption [354, 145] to encrypt in such a way that all decryption keys for that URI, *except for ones on a revocation list*, can be used to decrypt. Achieving this is nontrivial because we have to combine broadcast encryption with JEDI's semantics of hierarchical resources, expiry, and delegation. First, we modify broadcast encryption to support delegation, in such a way that if a key is revoked, all delegations made with that key are also implicitly revoked. Then, we integrate broadcast revocation, in a *non-black-box* way, with JEDI's encryption and delegation, as a third resource hierarchy alongside URIs and expiry. Our protocol for immediate revocation is not described in this dissertation; see our extended paper [297] for details.

7.1.3 Summary of Evaluation

For our evaluation, we use JEDI to encrypt messages transmitted over bw2 [16, 94], a deployed open-source messaging system for smart buildings, and demonstrate that JEDI's overhead is small in the critical path. We also evaluate JEDI for a commercially available sensor platform called "Hamilton" [210], and show that a Hamilton-based sensor sending one sensor reading every 30 seconds would see several years of battery lifetime when sending sensor readings encrypted with JEDI. As Hamilton is among the least powerful platforms that will participate in IoT (farthest to the right in Figure 7.1), this validates that JEDI is practical across the IoT spectrum.

7.2 System Model and Threat Model

A principal can post a message to a resource in a hierarchy by encrypting it according to the resource's URI, hierarchy's public parameters, and current time, and passing it to the underlying system that delivers it to the relevant subscribers. Given the secret key for a resource subtree and time range, a principal can generate a secret key for a subset of those resources and subrange of that time range, and give it to another principal, as in Figure 7.2. The receiving principal can use the delegated key to decrypt messages that are posted to a resource in that subset at a time during that subrange.

JEDI does not require the structure of the resource hierarchy to be fixed in advance. In Figure 7.2, the campus facilities manager, when granting access to `buildingA/*` to the building man-

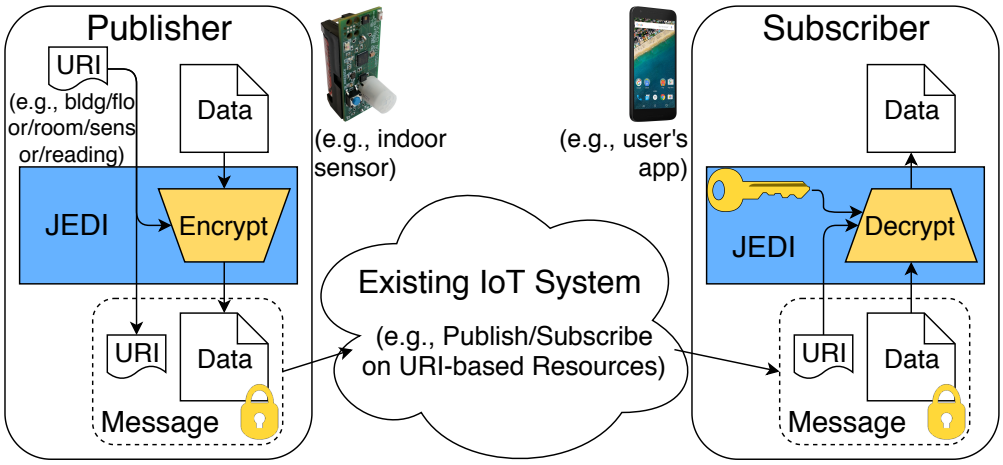


Figure 7.3: Applying JEDI to a smart buildings IoT system. Components introduced by JEDI are shaded. The subscriber's key is obtained via JEDI's decentralized delegation (Figure 7.2).

ager, need not be concerned with the structure of the subtree rooted at `buildingA`. This allows the building manager to organize `buildingA/*` independently.

7.2.1 Trust Assumptions

A principal is trusted for the resources it owns or was given access to (for the time ranges for which it was given access). In other words, an adversary who compromises a principal can read all resources that principal can read and forge new messages as if it were that principal. In particular, an adversary who compromises the authority for a resource hierarchy gains control over that resource hierarchy.

JEDI allows each principal to act as an authority for its own resource hierarchy in its own trust domain, without a single authority spanning all hierarchies. In particular, *principals* are not organized hierarchically; a principal may be delegated multiple keys, each belonging to a different resource hierarchy. In the example in Figure 7.2, Alice might also receive JEDI keys from her landlord granting access to resources in her apartment building, in a separate hierarchy where her landlord is the authority. If Alice owns resources she would like to delegate to others, she can set up her own hierarchy to represent those resources. Existing IoT systems with decentralized delegation, like bw2 and Vanadium, use a similar model.

7.2.2 Applying JEDI to an Existing System

As shown in Figure 7.3, JEDI can be applied as a wrapper around existing many-to-many communication systems, including publish-subscribe systems for smart cities. The transfer of messages from producers to consumers is handled by the existing system. A common design used by such systems is to have a central broker (or router) forward messages; however, an adversary who com-

promises the broker can read all messages. In this context, JEDI's end-to-end encryption protects data from such an adversary. Publishers encrypt their messages with JEDI before passing them to the underlying communication system (without knowledge of who the subscribers are), and subscribers decrypt them with JEDI after receiving them from the underlying communication system (without knowledge of who the publishers are).

7.2.3 Comparison to a Naïve Key Server Model

To better understand the benefits of JEDI's model, consider the natural strawman of a trusted key server. This key server generates a key for every URI and time. A publisher encrypts each message for that URI with the same key. A subscriber requests this key from the trusted key server, which must first check if the subscriber is authorized to receive it. The subscriber can decrypt messages for a URI using this key, and contact the key server for a new key when the key expires. JEDI's model is better than this key server model as follows:

- *Improved security.* Unlike the trusted key server, which must always be online, the authority in JEDI can delegate qualified keys to some principals *and then go offline*, leaving these principals to qualify and delegate keys further. While the authority is offline, it is more difficult for an attacker to compromise it and easier for the authority to protect its secrets because it need only access them rarely. This reasoning is the basis of root Certificate Authorities (CAs), which access their master keys infrequently. In contrast, the trusted key server model requires a central trusted party (key server) to be online to grant/revoke access to any resource.

- *Improved privacy.* No single participant sees all delegations in JEDI. An adversary in JEDI who steals an authority's secret key can decrypt all messages for that hierarchy, but still does not learn who has access to which resource, and cannot access separate hierarchies to which the first authority has no access. In contrast, an adversary who compromises the key server learns who has access to which resource and can decrypt messages for all hierarchies.

- *Improved scalability.* In the campus IoT example above, if a building admin receives access to all sensors and all their different readings for a building, the admin must obtain a potentially very large number of keys, instead of one key for the entire building. Moreover, the campus-wide key server needs to grant decryption keys to each application owned by each employee or student at the university. Finally, the campus-wide key server must understand which delegations are allowed at lower levels in the hierarchy, requiring the entire hierarchy to be centrally administered.

7.2.4 IoT Gateways

Low-power wireless embedded sensors, due to power constraints, often do not use network protocols like Wi-Fi, and instead use specialized low-power protocols such as Bluetooth or IEEE 802.15.4. It is common for these devices to rely on an *application-layer gateway* to send data to

computers outside of the low-power network [502]. This gateway could be in the form of a phone app (e.g., Fitbit), or in the form of a specialized border router [517, 487]. In some traditional setups, the gateway is responsible for performing encryption/authentication [375]. JEDI accepts that gateways may be necessary for Internet connectivity, but does not rely on them for security—JEDI's cryptography is lightweight enough to run directly on the low-power sensor nodes. This approach prevents the gateway from becoming a single point of attack; an attacker who compromises the gateway cannot see or forge data for any device using that gateway.

7.2.5 Generalizability of JEDI's Model

Since JEDI decouples senders from receivers, it has no requirements on what happens at any intermediaries (e.g., does not require messages to be forwarded from publishers to subscribers in any particular way). Thus, JEDI works even when messages are exchanged in a broadcast medium, e.g., multicast. This also means that JEDI is more broadly applicable to systems with hierarchically organized resources. For example, URIs could correspond to filepaths in a file system, or URLs in a RESTful web service.

7.2.6 Security Goals

JEDI's goal is to ensure that principals can only read messages from or send messages to resources they have been granted access to receive from or send to. In the context of publish-subscribe, JEDI also hides the content of messages from an adversary who controls the router.

JEDI does not attempt to hide metadata relating to the actual transfer of messages (e.g., the URIs on which messages are published, which principals are publishing or subscribing to which resources, and timing). Hiding this metadata is a complementary task to achieving delegation and end-to-end encryption in JEDI, and techniques from the secure messaging literature [113, 128, 222] will likely be applicable.

7.3 End-to-End Encryption

A central question answered in this section is: How should publishers encrypt messages before passing them to the underlying system for delivery (Section 7.3.4)? As explained in Section 7.1.2.2, ABE, the obvious choice, is too heavy for low-power devices. Therefore, we apply the technique from Section 3.2.3 and identify WKD-IBE, a more lightweight identity-based encryption scheme, as sufficient to achieve JEDI's properties. The primary challenge is to encode a sufficiently expressive rendezvous point in the WKD-IBE ID (called a *pattern*) that publishers use to encrypt messages (Section 7.3.4).

7.3.1 Building Block: WKD-IBE

We first explain WKD-IBE [2], the encryption scheme that JEDI uses as a building block. We denote the security parameter as κ.

7.3.1.1 WKD-IBE Scheme

In WKD-IBE, messages are encrypted with *patterns*, and keys also correspond to patterns. A pattern is a list of values: $P = (\mathbb{Z}_p^* \cup \{\perp\})^\ell$. The notation $P(i)$ denotes the ith component of P, 1-indexed. A pattern P_1 *matches* a pattern P_2 if, for all $i \in [1,\ell]$, either $P_1(i) = \perp$ or $P_1(i) = P_2(i)$. In other words, if P_1 specifies a value for an index i, P_2 must match it at i. Note that the "matches" operation is not commutative; "P_1 matches P_2" does not imply "P_2 matches P_1".

We refer to a component of a pattern containing an element of $\mathbb{Z}_p^*$ as *fixed*, and to a component that contains $\perp$ as *free*. To aid our presentation, we define the following sets:

Definition 1. *For a pattern S, we define:*

$$\mathit{fixed}(S) = \{(i, S(i)) \mid S(i) \neq \perp\}$$
$$\mathit{free}(S) = \{i \mid S(i) = \perp\}$$

A key for pattern P_1 can decrypt a message encrypted with pattern P_2 if $P_1 = P_2$. Furthermore, a key for pattern P_1 can be used to derive a key for pattern P_2, as long as P_1 matches P_2. In summary, the following is the syntax for WKD-IBE.

- **Setup**$(1^\kappa, 1^\ell) \to \mathsf{Params}, \mathsf{MasterKey}$;
- **KeyDer**$(\mathsf{Params}, \mathsf{Key}_{\mathsf{Pattern}_\mathsf{A}}, \mathsf{Pattern}_\mathsf{B}) \to \mathsf{Key}_{\mathsf{Pattern}_\mathsf{B}}$, derives a key for $\mathsf{Pattern}_\mathsf{B}$, where either $\mathsf{Key}_{\mathsf{Pattern}_\mathsf{A}}$ is the $\mathsf{MasterKey}$, or $\mathsf{Pattern}_\mathsf{A}$ matches $\mathsf{Pattern}_\mathsf{B}$;
- **Encrypt**$(\mathsf{Params}, \mathsf{Pattern}, m) \to \mathsf{Ciphertext}_{\mathsf{Pattern},m}$;
- **Decrypt**$(\mathsf{Key}_{\mathsf{Pattern}}, \mathsf{Ciphertext}_{\mathsf{Pattern},m}) \to m$.

We use the WKD-IBE construction in Section 3.2 of [2], based on BBG HIBE [63]. Like the BBG construction, it has constant-size ciphertexts, but requires the maximum pattern length ℓ to be known at Setup time. In this WKD-IBE construction, patterns containing $\perp$ can only be used in **KeyDer**, not in **Encrypt**; we extend it to support encryption with patterns containing $\perp$.

7.3.1.2 WKD-IBE Construction

We present the WKD-IBE construction in Section 3.2 of [2], including our extension to support encryption with patterns containing $\perp$.

The construction is based on bilinear groups. $\mathbb{G}$ and $\mathbb{G}_T$ are cyclic groups of prime order p, and they are related by a bilinear map $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$. The security parameter κ is related to the number of bits of p. We implemented WKD-IBE using BLS12-381, an *asymmetric* bilinear group

whose bilinear map is of the form $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$. We denote $\mathbb{G}_1$ the smaller and faster of the two source groups (elliptic curve over $\mathbb{F}_p$). The construction of WKD-IBE was originally defined for symmetric bilinear groups. Our description below shows how we mapped the construction onto an asymmetric bilinear group.

Setup(1^ℓ): Select $g \xleftarrow{\$} \mathbb{G}_2$ and $g_2, g_3, h_1, \ldots, h_\ell, h_s \xleftarrow{\$} \mathbb{G}_1$. Then select $\alpha \xleftarrow{\$} \mathbb{Z}_p$ and let $g_1 = g^\alpha$. Output:
$\mathsf{Params} = (g, g_1, g_2, g_3, h_1, \ldots, h_\ell, h_s)$ and $\mathsf{MasterKey} = g_2^\alpha$.
Note that, although h_s is not used for encryption below, h_s will be used in Section 7.4.

KeyDer(K, S): If K is the master key, take $K = g_2^\alpha$. Select $r \xleftarrow{\$} \mathbb{Z}_p$. The private key for the pattern S is the following triple:

$$\left(g_2^\alpha \cdot \left(g_3 \cdot \prod_{(i,a_i) \in \text{fixed}(S)} h_i^{a_i} \right)^r, \quad g^r, \quad \left\{ (j, h_j^r) \right\}_{j \in \text{free}(S)} \right).$$

If K is not the master key, then parse K as (k_0, k_1, B), where $B = \{(i, b_i)\}$. Select $t \xleftarrow{\$} \mathbb{Z}_p$. The private key for S is:

$$\left(k_0 \cdot \left(g_3 \cdot \prod_{(i,a_i) \in \text{fixed}(S)} h_i^{a_i} \right)^t \cdot \prod_{\substack{(i,a_i) \in \text{fixed}(S) \\ (i,b_i) \in B}} b_i^{a_i}, \quad g^t \cdot k_1, \quad \left\{ (j, h_j^t \cdot b_j) \right\}_{j \in \text{free}(S)} \right).$$

Observe that the resulting key is identically distributed, regardless of whether or not the input key K is the master key.

Encrypt(S, m): Here, $m \in \mathbb{G}_T$. Select $s \xleftarrow{\$} \mathbb{Z}_p$ and output

$$\left(e(g_1, g_2)^s \cdot m, \quad g^s, \quad \left(g_3 \cdot \prod_{(i,a_i) \in \text{fixed}(S)} h_i^{a_i} \right)^s \right).$$

Decrypt(K, C): Parse the key K as (k_0, k_1, B), and the ciphertext C as (X, Y, Z). Output

$$X \cdot e(k_1, Z) \cdot e(Y, k_0)^{-1}.$$

To support encryption over arbitrary patterns that may contain $\perp$, we only compute the product over fixed slots, just as is done in the BBG HIBE construction [63]. In contrast, the original WKD-IBE construction requires all slots in the pattern to be fixed, and iterates over all slots. The proof technique from [63], namely padding the selected ID with zeros, can be used here to modify the proof of WKD-IBE [2] to account for our optimization that allows free slots to be used in encryption.

7.3.2 Concurrent Hierarchies in JEDI

WKD-IBE was originally designed to allow delegation in a *single* hierarchy. For example, the original suggested use case of WKD-IBE was to generate secret keys for a user's email addresses in all valid subdomains, such as `sysadmin@*.univ.edu` [2].

JEDI, however, uses WKD-IBE in a nonstandard way to simultaneously support *multiple* hierarchies, one for URIs and one for expiry (and, as explained in the extended paper [297], one for revocation), each in the vein of HIBE. We think of the ℓ components of a WKD-IBE pattern as "slots" that are initially empty, and are progressively filled in with calls to **KeyDer**. To combine a hierarchy of maximum depth ℓ_1 (e.g., the URI hierarchy) and a hierarchy of maximum depth ℓ_2 (e.g., the expiry hierarchy), one can **Setup** WKD-IBE with the number of slots equal to $\ell = \ell_1 + \ell_2$. The first ℓ_1 slots are filled in left-to-right for the first hierarchy and the remaining ℓ_2 slots are filled in left-to-right for the second hierarchy (Figure 7.4).

7.3.3 Overview of Encryption in JEDI

Each principal maintains a **key store** containing WKD-IBE decryption keys. To create a resource hierarchy, any principal can call the WKD-IBE **Setup** function to create a resource hierarchy. It releases the *public parameters* and stores the *master secret key* in its key store, making it the authority of that hierarchy. To delegate access to a URI prefix for a time range, a principal (possibly the authority) searches its key store for a set of keys for a superset of those permissions. It then qualifies those keys using **KeyDer** to restrict them to the specific URI prefix and time range (Section 7.3.5), and sends the resulting keys to the recipient of the delegation.[2] The recipient accepts the delegation by adding the keys to its key store.

Before sending a message to a URI, a principal encrypts the message using WKD-IBE. The pattern used to encrypt it is derived from the URI and the current time (Section 7.3.4), which are included along with the ciphertext. When a principal receives a message, it searches its key store, using the URI and time included with the ciphertext, for a key to decrypt it.

In summary, JEDI provides the following API:

Encrypt(Message, URI, Time) $\rightarrow$ Ciphertext

Decrypt(Ciphertext, URI, Time, KeyStore) $\rightarrow$ Message

Delegate(KeyStore, URIPrefix, TimeRange) $\rightarrow$ KeySet

AcceptDelegation(KeyStore, KeySet) $\rightarrow$ KeyStore$'$

Note that the WKD-IBE public parameters are an implicit argument to each of these functions. Finally, although the above API lists the arguments to **Delegate** as URIPrefix and TimeRange, JEDI actually supports succinct delegation over more complex sets of URIs and timestamps (see Section 7.3.10).

[2] JEDI does not govern *how* the key set is transferred to the recipient, as there are existing solutions for this. One can use an existing protocol for one-to-one communication (e.g., TLS) to securely transfer the key set. Or, one can encrypt the key set with the recipient's (normal, non-WKD-IBE) public key, and place it in a common storage area.

$S(i)$	H("a")	H("b")	H($)	$\perp$	H("17")	H("Jun")	H("08")	H("06")
i	1	2	3	4	5	6	7	8

$\ell_1 = 4$ slots for URI Hierarchy $\quad$ $\ell_2 = 4$ slots for Time Hierarchy

Figure 7.4: Pattern S used to encrypt message sent to a/b on June 08, 2017 at 6 AM. The figure uses 8 slots for space reasons; JEDI is meant to be used with more slots (e.g., 20).

7.3.4 Expressing URI/Time as a Pattern

A message is encrypted using a pattern derived from (1) the URI to which the message is addressed, and (2) the current time. Let $H : \{0,1\}^* \to \mathbb{Z}_p^*$ be a collision-resistant hash function. Let $\ell = \ell_1 + \ell_2$ be the pattern length in the hierarchy's WKD-IBE system. We use the first ℓ_1 slots to encode the URI, and the last ℓ_2 slots to encode the time.

Given a URI of length d, such as a/b/c ($d = 3$ in this example), we split it up into individual components, and append a special terminator symbol $: ("a", "b", "c", $). Using H, we map each component to $\mathbb{Z}_p^*$, and then put these values into the first $d+1$ slots. If S is our pattern, we would have $S(1) = H$("a"), $S(2) = H$("b"), $S(3) = H$("c"), and $S(4) = H$($) for this example. Now, we encode the time range into the remaining ℓ_2 slots. Any timestamp, with the granularity of an hour, can be represented hierarchically as (year, month, day, hour). We encode this into the pattern like the URI: we hash each component, and assign them to consecutive slots. The final ℓ_2 slots encode the time, so the depth of the time hierarchy is ℓ_2. The terminator symbol $ is not needed to encode the time, because timestamps always have exactly ℓ_2 components. For example, suppose that a principal sends a message to a/b on June 8, 2017 at 6 AM. The message is encrypted with the pattern in Figure 7.4.

7.3.5 Producing a Key Set for Delegation

Now, we explain how to produce a key set corresponding to a URI prefix and time range. To express a URI prefix as a pattern, we do the same thing as we did for URIs, without the terminator symbol $. For example, a/b/* is encoded in a pattern S as $S(1) = H$("a"), $S(2) = H$("b"), and all other slots free. Given the private key for S, one can use WKD-IBE's **KeyDer** to fill in slots $3 \ldots \ell_1$. This allows one to generate the private key for a/b, a/b/c, etc.—any URI for which a/b is a prefix. To grant access to only a specific resource (a full URI, not a prefix), the $ is included as before.

In encoding a time range into a pattern, single timestamps (e.g., granting access for an hour) are done as before. The hierarchical structure for time makes it possible to succinctly grant permission for an entire day, month, or year. For example, one may grant access for all of 2017 by filling in slot ℓ_2 with H("2017") and leaving the final $\ell_2 - 1$ slots, which correspond to month, day, and year, free. Therefore, to grant permission over a time range, *the number of keys granted is logarithmic in the length of the time range*. For example, to delegate access to a URI from October 29, 2014

at 10 PM until December 2, 2014 at 1 AM, the following keys need to be generated: `2014/Oct/29/23`, `2014/Oct/29/24`, `2014/Oct/30/*`, `2014/Oct/31/*`, `2014/Nov/*`, `2014/Dec/01/*`, and `2014/Dec/02/01`. The tree can be chosen differently to support longer time ranges (e.g., additional level representing decades), change the granularity of expiry (e.g., minutes instead of hours), trade off encryption time for key size (e.g., deeper/shallower tree), or use a more regular structure (e.g., binary encoding with logarithmic split). For example, our implementation uses a depth-6 tree (instead of depth-4), to be able to delegate time ranges with fewer keys.

In summary, to produce a key set for delegation, first determine which subtrees in the time hierarchy represent the time range. For each one, produce a separate pattern, and encode the time into the last ℓ_2 slots. Encode the URI prefix in the first ℓ_1 slots of each pattern. Finally, generate the keys corresponding to those patterns, using keys in the key store.

7.3.6 Using WKD-IBE Efficiently

On low-power embedded devices, performing a single WKD-IBE encryption consumes a significant amount of energy. Therefore, we design JEDI to use WKD-IBE as efficiently as possible.

7.3.6.1 Hybrid Encryption and Key Reuse

We apply the technique from Section 3.2.1, using WKD-IBE in a hybrid encryption scheme. To encrypt a message m in JEDI, one samples a symmetric key k, and encrypts k with JEDI to produce ciphertext c_1. The pattern used for WKD-IBE encryption is chosen as in Section 7.3.4 to encode the *rendezvous point*. Then, one encrypts m using k to produce ciphertext c_2. The JEDI ciphertext is (c_1, c_2).

For subsequent messages, one reuses k and c_1; the new message is encrypted with k to produce a new c_2. One can keep reusing k and c_1 until the WKD-IBE pattern for encryption changes, which happens at the end of each hour (or other interval used for expiry). At this time, JEDI performs *key rotation* by choosing a new k, encrypting it with WKD-IBE using the new pattern, and then proceeding as before. Therefore, *most messages only incur cheap symmetric-key encryption*.

This also reduces the load on subscribers. The JEDI ciphertexts sent by a publisher during a single hour will all share the same c_1. Therefore, the subscriber can decrypt c_1 once for the first message to obtain k, and *cache* the mapping from c_1 to k to avoid expensive WKD-IBE decryptions for future messages sent during that hour.

Thus, expensive WKD-IBE operations are only performed upon key rotation. This not only happens *rarely* for each resource—for example, once an hour, if expiry times are encoded with hour-level granularity—but its frequency is tunable according to the granularity chosen for expiry. This allows JEDI to be configured to strike the best balance between the value that fine-grained expiry may bring to the application and the costs of invoking WKD-IBE more frequently. Therefore, using hybrid encryption in this way is an application of both the technique from Section 3.2.1 and the technique from Section 3.2.2.

7.3.6.2 Precomputation with Adjustment

Even with hybrid encryption and key reuse to perform WKD-IBE encryption rarely, WKD-IBE contributes significantly to the overall power consumption on low-power devices. Therefore, this section explores how to perform individual WKD-IBE encryptions more efficiently. We apply the technique from Section 3.2.4 to develop a new encryption algorithm for WKD-IBE that is well-suited to the way in which JEDI uses WKD-IBE.

Most of the work to encrypt a message under a pattern S is in computing the quantity $Q_S = g_3 \cdot \prod_{(i,a_i)\in\text{fixed}(S)} h_i^{a_i}$, where g_3 and the h_i are part of the WKD-IBE public parameters. One may consider computing Q_S once, and then reusing its value when computing future encryptions under the same pattern S. Unfortunately, this alone does not improve efficiency because the pattern S used in one WKD-IBE encryption is different from the pattern T used for the next encryption.

JEDI, however, observes that S and T are similar; they match in the ℓ_1 slots corresponding to the URI, and the remaining ℓ_2 slots will correspond to adjacent leaves in the time tree. JEDI takes advantage of this by efficiently *adjusting* the precomputed value Q_S to compute Q_T. We define the new WKD-IBE operations as follows (as before, Params is an implicit parameter).

Precompute(S): Output

$$g_3 \cdot \prod_{(i,a_i)\in\text{fixed}(S)} h_i^{a_i}$$

AdjustPrecomputed(Q_S, S, T): Q_S is the existing precomputed value, S is the pattern it corresponds to, and T is the pattern whose precomputed value Q_T to compute. Output

$$Q_S \cdot \prod_{\substack{(i,b_i)\in\text{fixed}(T)\\ i\in\text{free}(S)}} h_i^{b_i} \cdot \prod_{\substack{(i,a_i)\in\text{fixed}(S)\\ i\in\text{free}(T)}} h_i^{-a_i} \cdot \prod_{\substack{(i,a_i)\in\text{fixed}(S)\\ (i,b_i)\in\text{fixed}(T)\\ a_i\neq b_i}} h_i^{b_i-a_i}$$

EncryptPrepared(Q_S, m): Here, $m \in \mathbb{G}_T$. Select $s \xleftarrow{\$} \mathbb{Z}_p$ and output

$$(e(g_1,g_2)^s \cdot m, \quad g^s, \quad Q_S^s).$$

The **AdjustPrecomputed** operation requires one $\mathbb{G}_1$ exponentiation per differing slot between S and T (i.e., the Hamming distance). Because S and T usually differ in only the final slot of the time hierarchy, this will usually require one $\mathbb{G}_1$ exponentiation total, substantially faster than computing Q_T from scratch. Additional exponentiations are needed at the end of each day, month, and year, but they can be eliminated by maintaining additional precomputed values corresponding to the start of the current day, current month, and current year.

Concretely, JEDI encrypts a new symmetric key when rotating keys as follows. First, we invoke **AdjustPrecomputed** to compute Q_S for the desired attribute set S from $Q_{S'}$, where S' is the previous attribute set. Then, we invoke **EncryptPrepared** to encrypt the symmetric key under the attribute set S using Q_S. We can represent this composition of **EncryptPrepared** and **AdjustPrecomputed** as a new encryption interface, **Encrypt**$(Q_{S'}, S', S, m) \to C, Q_S$. Here, encryption outputs not only a ciphertext C but also state Q_S to accelerate the next encryption, allowing encryption operations to be "chained together."

The protocol remains secure because a ciphertext is distributed identically whether it was computed from a precomputed value Q_S or via regular encryption—that is, the output of **Encrypt**(S, m) and the output of **EncryptPrepared**(**Precompute**$(S), m)$ are distributed identically.

7.3.7 Revocation

In this section, we briefly explain below how JEDI keys may be revoked.

7.3.8 Simple Solution: Revocation via Expiry

A simple solution for revocation is to rely on expiration. In this solution, all keys are time-limited, and delegations are periodically refreshed, according to a higher layer protocol, by granting a new key with a later expiry time. In this setup, the principal who granted a key can easily revoke it by not refreshing that delegation when the key expires. We expect this solution to be sufficient for many applications of JEDI.

7.3.9 Immediate Revocation (Extended Paper)

Some disadvantages of the solution in Section 7.3.8 are that (1) principals must periodically come online to refresh delegations, and (2) revocation only takes effect when the delegated key expires. We would like a solution without these disadvantages.

However, any revocation scheme that does not wait for keys to expire is subject to set of *inherent* limitations. The recipient of the revoked delegation still has the revoked decryption key, so it can still decrypt messages encrypted in the same way. This means that we must either (1) rely on intermediate parties to modify ciphertexts so that revoked keys cannot decrypt them, or (2) require senders to be aware of the revocation, and encrypt messages in a different way so that revoked keys cannot decrypt them. Neither solution is ideal: (1) makes assumptions about how messages are delivered, which we have avoided thus far (Section 7.2), and requires trust in an intermediary to modify ciphertexts, and (2) weakens the decoupling of senders and receivers (Section 7.1.2). In our extended JEDI paper [297], we present an immediate revocation scheme that adopts the second compromise: while senders will not need to know who are the receivers, they will need to know who has been revoked.

7.3.10 Extensions

We present two simple extensions to JEDI's core encryption protocol: (1) generalized subtrees with wildcards in the *middle* of a URI, and (2) forward secrecy.

7.3.10.1 Beyond Simple Hierarchies

Thus far, we have considered only the $*$ wildcard at the end of a URI. With WKD-IBE, we can also place a + wildcard in the middle of a URI, allowing a single component of the URI to remain

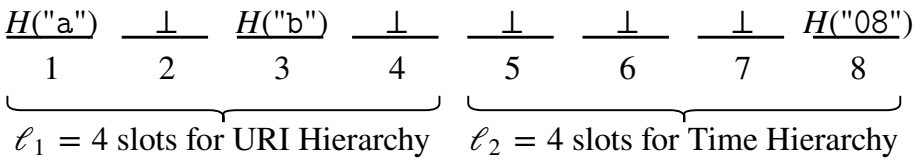


Figure 7.5: Pattern S for a private key granting access to `a/+/b/*` at 8 AM every day. The figure uses 8 slots for space reasons; JEDI is meant to be used with more slots (e.g., 20).

unspecified. For example, the URI `a/+/b` matches all URIs of length 3 where the first component is `a` and the third component is `b`; the second component could be anything. To implement the `+` wildcard, we fill in the components corresponding to `+` with $\perp$.

The `+` wildcard is useful in real applications. For example, in the "smart buildings" setting, one could imagine a resource hierarchy of the form `buildingA/floor2/room/sensor_id/reading_type`, where `reading_type` could be either `temp` or `hum`. The `+` wildcard allows one to delegate permission to see only the temperature readings in a building, by granting permission on the URI `buildingA/+/+/+/temp`. It is also useful for the time hierarchy. An organization may want to give an employee access to a resource from 8 AM to 5 PM every day, which can be accomplished by using the `+` wildcard for the slots corresponding to the year, month, and day. See Figure 7.5.

7.3.10.2 Forward Secrecy

Forward secrecy is the property that if a subscriber's decryption key is compromised, the attacker should only be able to decrypt *new* messages visible to the subscriber, not old messages sent before the key was compromised.

Forward secrecy using HIBE has been previously studied [97]. We can apply the same idea to our construction via a straightforward extension to our mechanism for expiry. In our construction of expiry, each subscriber has a collection of keys for each URI or URI prefix it can access that give it access over a time range $[t_1, t_2]$, which can be qualified to any smaller time range $[t_3, t_4]$ where $t_1 \leq t_3 \leq t_4 \leq t_2$. To achieve forward secrecy, each subscriber qualifies the keys for each URI, at each unit of time, to only be valid starting at the *current time* until the same expiry time, and then discards the old keys. This guarantees that, if a key is stolen, it cannot be used to decrypt messages published before the current time.

7.3.11 Security Guarantee

In this section, we present our formal definitions of JEDI's security guarantees and the corresponding proofs. Our proofs use the notion of IND-sWKID-CPA security defined in Section 3 of [2]. They also depend on a property of the construction of WKD-IBE called *history-independence*. Informally, a WKD-IBE construction is *history-independent* if, for any fixed pattern S, the result of **KeyDer** to produce a key with pattern S, assuming that the starting key is either the master key

or corresponds to a pattern that matches S, is distributed in exactly the same way regardless of the particular starting key used. The idea is that, given a key for a specified pattern S, one learns nothing about the sequence of **KeyDer** operations that produced the key.

We formally define history-independence below.

Definition 2 (History-Independence)**.** *A WKD-IBE construction is said to be history-independent if, for every pattern S, and for any two well-formed keys k_1 corresponding to pattern P_1 and k_2 corresponding to pattern P_2 in the same WKD-IBE system such that P_1 matches S and P_2 matches S, it holds that*

$$\{\mathbf{KeyDer}(k_1, S)\} = \{\mathbf{KeyDer}(k_2, S)\}$$

where the distributions are over the randomness sampled internally by **KeyDer**. *If k_1 (respectively, k_2) is the master key, the pattern P_1 (respectively, P_2) is one where all slots are free.*

Below, we show that the construction of WKD-IBE presented in Section 7.3.1.2, which JEDI uses, satisfies this property.

Theorem 1. *The construction of WKD-IBE presented in Section 7.3.1.2 is history-independent.*

Proof of Theorem 1. We will show that for any pattern S and any well-formed key k corresponding to a pattern P that matches S, it holds that

$$\{\mathbf{KeyDer}(k, S)\} = \left\{ \left(g_2^{\alpha} \cdot \left(g_3 \cdot \prod_{(i,a_i)\in \text{fixed}(S)} h_i^{a_i} \right)^{r},\ g^{r},\ \left\{ (j, h_j^{r}) \right\}_{j\in\text{free}(S)} \right) \right\}_{r \xleftarrow{\$} \mathbb{Z}_p}$$

Because the formula on the right-hand side of the above equation only depends on S and the public parameters (not the particular key k), this is sufficient to demonstrate history-independence of the WKD-IBE construction.

We handle the proof in two cases.

Case 1. Suppose that k is the master key. Then the above result is true by definition, according to the formula given for **KeyDer** in Section 7.3.1.2.

Case 2. Suppose that k is not the master key. Then, because k is well-formed, we can write that

$$k = \left(g_2^{\alpha} \cdot \left(g_3 \cdot \prod_{(i,a_i)\in \text{fixed}(P)} h_i^{a_i} \right)^{r_0},\ g^{r_0},\ \left\{ (j, h_j^{r_0}) \right\}_{j\in\text{free}(P)} \right)$$

for some fixed $r_0 \in \mathbb{Z}_p$. By applying the formula for **KeyDer** in Section 7.3.1.2, we can see that the key output by **KeyDer** has the form

$$\left(g_2^{\alpha} \cdot \left(g_3 \cdot \prod_{(i,a_i)\in \text{fixed}(S)} h_i^{a_i} \right)^{r_0+t},\ g^{r_0+t},\ \left\{ (j, h_j^{r_0+t}) \right\}_{j\in\text{free}(S)} \right)$$

for $t \xleftarrow{\$} \mathbb{Z}_p$. Because $r_0 + t$ is uniformly distributed in $\mathbb{Z}_p$, the output key has the desired distribution (take $r = r_0 + t$). □

Now, we are ready to state and prove a security guaranatee for JEDI's encryption. We formalize the security of JEDI's encryption below.

Theorem 2. *Suppose JEDI is instantiated with a WKD-IBE scheme that is Selective-ID CPA-secure [62, 2] and history-independent. Then, no probabilistic polynomial-time adversary $\mathcal{A}$ can win the following security game against a challenger $\mathcal{C}$ with non-negligible advantage:*
Initialization. *$\mathcal{A}$ selects a (URI, time) pair to attack.*
Setup. *$\mathcal{C}$ gives $\mathcal{A}$ the public parameters of the JEDI instance.*
Phase 1. *$\mathcal{A}$ can make three types of queries to $\mathcal{C}$:*

1. *$\mathcal{A}$ asks $\mathcal{C}$ to create a principal; $\mathcal{C}$ returns a name in $\{0,1\}^*$, which $\mathcal{A}$ can use to refer to that principal in future queries. A special name exists for the authority.*
2. *$\mathcal{A}$ asks $\mathcal{C}$ for the key set of any principal; $\mathcal{C}$ gives $\mathcal{A}$ the keys that the principal has. At the time this query is made, the requested key may **not** contain a key whose URI and time are both prefixes of the challenge (URI, time) pair.*
3. *$\mathcal{A}$ asks $\mathcal{C}$ to make any principal delegate a key set of $\mathcal{A}$'s choice to another principal (specified by names in $\{0,1\}^*$).*

Challenge. *When $\mathcal{A}$ chooses to end Phase 1, it sends $\mathcal{C}$ two messages, m_0 and m_1, of the same length. Then $\mathcal{C}$ chooses a random bit $b \in \{0,1\}$, encrypts m_b under the challenge (URI, time) pair, and gives $\mathcal{A}$ the ciphertext.*
Phase 2. *$\mathcal{A}$ can make additional queries as in Phase 1.*
Guess. *$\mathcal{A}$ outputs $b' \in \{0,1\}$, and wins the game if $b = b'$. The advantage of an adversary $\mathcal{A}$ is $\left|\Pr[\mathcal{A}\ wins] - \frac{1}{2}\right|$.*

Now, we prove Theorem 2. Some intuition behind the proof is that the challenger in the game in Theorem 2 (which is also the adversary in the IND-sWKID-CPA game [2]), maintains, for each principal, the set of keys it has. It *lazily* requests these keys from the IND-sWKID-CPA challenger as principals are compromised. Therefore, it maintains (1) a *key set* for each principal, storing the keys requested from the IND-sWKID-CPA challenger, and (2) a *pattern set* for each principal, storing patterns corresponding to additional keys that the principal would have in the normal course of JEDI, but which have not been requested from the IND-sWKID-CPA challenger yet. Requesting keys lazily is crucial because an uncompromised principal in Theorem 2 may possess a secret key capable of decrypting the challenge ciphertext.

Proof of Theorem 2. We show that, given an adversary $\mathcal{A}$ with non-negligible advantage in the game in Theorem 2, one can construct an algorithm $\mathcal{B}$ with non-negligible advantage in the IND-sWKID-CPA security game [2]. We denote as $\mathcal{C}$ the IND-sWKID-CPA security challenger. $\mathcal{B}$ maintains the following state: (2) a mapping from principal name (in $\{0,1\}^*$) to a key set for that principal, and (3) a mapping from principal name (in $\{0,1\}^*$) to a pattern set for that principal. These two maps are initialized as follows; each has a single entry for the name corresponding to

the authority. The authority's key set is empty, and its pattern set contains one element, namely a pattern containing $\perp$ in all components (i.e., with all slots free).

$\mathcal{B}$ first runs the game with $\mathcal{A}$ as the challenger. $\mathcal{A}$ specifies the pair (URI, time) that it will attack at the beginning of the game. $\mathcal{B}$ parses (URI, time) into the pattern S^* and gives it to $\mathcal{C}$. $\mathcal{C}$ generates the master key pair $(\mathsf{mpk}, \mathsf{msk}) \leftarrow \mathsf{Setup}$ and gives B the master public key mpk. $\mathcal{B}$ forwards mpk to $\mathcal{A}$. For any of three queries from $\mathcal{A}$ in Phase 1, $\mathcal{B}$ processes it as following:

- $\mathcal{A}$ asks $\mathcal{B}$ to create a principal: $\mathcal{B}$ returns a fresh name in $\{0,1\}^*$ corresponding to the new principal. $\mathcal{B}$ creates mappings from this name to an empty set, for both the key set and pattern set, indicating that this new principal has not been delegated any keys.

- $\mathcal{A}$ asks $\mathcal{B}$ for the key set of a principal p: $\mathcal{B}$ finds in its local state the key set and pattern set for p. For each pattern in p's pattern set, it queries $\mathcal{A}$ for the corresponding WKD-IBE secret key. It adds each WKD-IBE secret key to p's key set, and then replaces p's pattern set in its local state with an empty set. Then it returns the keys in p's key set to $\mathcal{A}$. Note that $\mathcal{B}$ will not query $\mathcal{C}$ the secret key for a pattern that matches S^*, because, in the game in Theorem 2, $\mathcal{A}$ is not allowed to request a key set containing a key whose URI and time match the challenge pair (URI, time). Also note that the keys given to $\mathcal{A}$ are distributed exactly as they would be in the JEDI protocol, because the underlying WKD-IBE scheme is assumed to be history-independent.

- $\mathcal{A}$ asks an principal p to make a delegation of $\mathcal{A}$'s choice of another principal q: $\mathcal{B}$ finds in its local state the key set and pattern set for p. $\mathcal{B}$ obtains the pattern corresponding to each key in p's key set. Let M be the set containing those patterns. $\mathcal{B}$ computes the set N, which is the union of M and p's pattern set. Based on the patterns in N, $\mathcal{B}$ computes the patterns corresponding to the keys that p would generate and delegate to q. For each such key, $\mathcal{B}$ adds the corresponding pattern to q's pattern set.

At the end of Phase 1, $\mathcal{A}$ outputs two equal-length challenge messages m_0 and m_1, and sends them to $\mathcal{B}$. $\mathcal{B}$ then forwards m_0 and m_1 to $\mathcal{C}$. $\mathcal{C}$ chooses a random bit b, and sends $\mathcal{B}$ the ciphertext of m_b. $\mathcal{B}$ then forwards the ciphertext to $\mathcal{A}$.

In Phase 2, $\mathcal{A}$ makes additional queries as in Phase 1, and $\mathcal{C}$ can process them as before.

Finally, $\mathcal{A}$ will return the bit b'. $\mathcal{B}$ returns b' to $\mathcal{C}$. Because every response that $\mathcal{B}$ makes to $\mathcal{A}$ is distributed identically to the results of actually playing the game in Theorem 2, $\mathcal{A}$ will guess $b' = b$ with non-negligible advantage. Thus, $\mathcal{B}$ wins the IND-sWKID-CPA game with non-negligible advantage. □

We achieve selective security in the standard model, like much prior work [63, 2]. A natural question is how to achieve adaptive security. As has been observed for IBE [62], HIBE [63], and WKD-IBE [2], hashing each component of the ID results in adaptive security, but with a loss of security exponential in the size of the hash. However, if the hash function is modeled as a random oracle, and the number of slots in WKD-IBE is logarithmic in the security parameter, then the loss in security is polynomial [2] (assuming that the number of slots ℓ is logarithmic in the security

parameter). Given that we use a hash function to map URI/time to a pattern (Section 7.3.4), this analysis applies to JEDI.

It is sufficient for JEDI to use a CPA-secure (rather than CCA-secure) encryption scheme because JEDI messages are signed, as detailed below in Section 7.4.

7.4 Integrity

To prevent an attacker from flooding the system with messages, spoofing fake data, or actuating devices without permission, JEDI must ensure that a principal can only send a message on a URI if it has permission. For example, an application subscribed to `buildingA/floor2/roomLHall/sensor0/temp` should be able to verify that the readings it is receiving are produced by `sensor0`, not an attacker. In addition to subscribers, an intermediate party (e.g., the router in a publish-subscribe system) may use this mechanism to filter out malicious traffic, without being trusted to read messages.

7.4.1 Starting Solution: Signature Chains

A standard solution in the existing literature, used by SPKI/SDSI [121], Vanadium [444], and bw2 [16], is to include a certificate chain with each message. Just as permission to subscribe to a resource is granted via a chain of delegations in Section 7.3, permission to publish to a resource is also granted via a chain of delegations. Whereas Section 7.3 includes WKD-IBE keys in each delegation, these integrity solutions delegate signed certificates. To send a message, a principal encrypts it (Section 7.3), signs the ciphertext, and includes a certificate chain that proves that the signing keypair is authorized for that URI and time.

7.4.2 Anonymous Signatures

The above solution reveals the sender's identity (via its public key) and the particular chain of delegations that gives the sender access. For some applications this is acceptable, and its auditability may even be seen as a benefit. For other applications, the sender must be able to send a message anonymously. See Section 7.1.2.3 for an example. How can we reconcile *access control* (ensuring the sender has permission) and *anonymity* (hiding who the sender is)?

7.4.2.1 Starting Point: WKD-IBE Signatures

Our solution is to use a signature scheme based on WKD-IBE. Abdalla et al. [2] observe that WKD-IBE can be extended to a signature scheme in the same vein as has been done for IBE [65] and HIBE [186]. To sign a message $m \in \mathbb{Z}_p^*$ with a key for pattern S, one uses **KeyDer** to fill in a slot with m, and presents the decryption key as a signature.

This is our starting point for designing anonymous signatures in JEDI. A message can be signed by first hashing it to $\mathbb{Z}_p^*$ and signing the hash as above. Just as consumers receive decryption keys

via a chain of delegations (Section 7.3), publishers of data receive these signing keys via chains of delegations.

7.4.2.2 Anonymous Signatures in JEDI

The construction in Section 7.4.2.1 has two shortcomings. First, signatures are *large*, linear in the number of fixed slots of the pattern. Second, it is unclear if they are truly *anonymous*.

Signature size. As explained in Section 7.3, we use a construction of WKD-IBE based on BBG HIBE [63]. BBG HIBE supports a property called *limited delegation* in which a secret key can be reduced in size, in exchange for limiting the depth in the hierarchy at which subkeys can be generated from it. We observe that the WKD-IBE construction also supports this feature. Because we need not support **KeyDer** for the decryption key acting as a signature, we use limited delegation to compress the signature to just two group elements.

Signature verification. WKD-IBE signatures, as proposed in Section 7.4.2.1 are verified by encrypting a random message under the pattern corresponding to the signature, and then attempting to decrypt it using the key acting as a signature. We provide a more efficient signature verification algorithm for this construction of WKD-IBE, described below in Section 7.4.2.3.

Anonymity. The technique in Section 7.4.2.1 transforms an encryption scheme into a signature scheme, but the resulting signature scheme is not necessarily anonymous. For the particular construction of WKD-IBE that we use, however, we prove that the resulting signature scheme is indeed anonymous. Our insight is that, for this construction of WKD-IBE, keys are *history-independent* in the following sense: **KeyDer**, for a fixed Params and $\mathsf{Pattern}_\mathsf{B}$, returns a private key $\mathsf{Key}_{\mathsf{Pattern}_\mathsf{B}}$ with the *exact same distribution* regardless of $\mathsf{Key}_{\mathsf{Pattern}_\mathsf{A}}$ (see Section 7.3.1 for notation). Because signatures, as described in Section 7.4.2.1, are private keys generated with **KeyDer**, they are also history-independent; a signature for a pattern has the same distribution regardless of the key used to generate it. This is precisely the anonymity property we desire. In finding a way to use WKD-IBE for anonymous signatures, rather than a more expressive cryptographic scheme, we are applying the technique from Section 3.2.3.

7.4.2.3 Construction of WKD-IBE Signatures

We formally describe our signature algorithm below, based on the idea in Section 7.4.2.1 and including the optimizations in Section 7.4.2.2. Note that the term h_s in the public parameters and an analogous element (s, b_s) in the third component of each secret key represent the slot dedicated to signing messages. They were not present in the original WKD-IBE construction and they are not used for encryption or decryption in Section 7.3.1.2.

Sign(K, m): Parse the key K as (k_0, k_1, B), where $(s, b_s) \in B$. Let S be the pattern corresponding to K. Select $t \xleftarrow{\$} \mathbb{Z}_p$ and output

$$\left(k_0 \cdot \left(g_3 \cdot h_s^m \cdot \prod_{(i,a_i) \in \text{fixed}(S)} h_i^{a_i} \right)^t \cdot b_s^m, \quad g^t \cdot k_1 \right)$$

Verify(S,σ,m): Parse the signature σ as (s_0,s_1). Check:

$$e(s_0,g) \stackrel{?}{=} e(g_1,g_2)\cdot e\left(g_3\cdot h_s^m\cdot \prod_{(i,a_i)\in \text{fixed}(S)} h_i^{a_i},\quad s_1\right)$$

In contrast to optimized procedures above, the naïve signature algorithm has **Sign**$(K,m) =$ **KeyDer**(K,T), and **Verify**$(S,\sigma,m) =$ (**Decrypt**$(\sigma,$**Encrypt**$(T,m^*)) \stackrel{?}{=} m^*)$ for $m^* \stackrel{\$}{\leftarrow} \mathbb{Z}_p^*$, where T is the same as S except that $T(s) = m$, the message being signed. The modification we make is that (1) the signature contains only the first two components of **KeyDer**(K,T) (since the third component is not used for decryption), and (2) the verification procedure checks that σ is a private key corresponding to T more efficiently than encrypting and decrypting a random message.

Finally, note that the **Sign** function can be generalized to allow a key with pattern P to produce a signature for pattern S if P matches S. This can be done trivially by first applying **KeyDer** to obtain a key for S, and calling the **Sign** on the existing key. Our implementation supports this **GeneralizedSign** functionality more efficiently, as follows:

GeneralizedSign(K,S,m): Parse the key K as (k_0,k_1,B), where $(s,b_s) \in B$. Select $t \stackrel{\$}{\leftarrow} \mathbb{Z}_p$ and output

$$\left(k_0\cdot\left(g_3\cdot h_s^m\cdot \prod_{(i,a_i)\in \text{fixed}(S)} h_i^{a_i}\right)^t \cdot b_s^m \cdot \prod_{\substack{(i,a_i)\in \text{fixed}(S)\\(i,b_i)\in B}} b_i^{a_i},\quad g^t\cdot k_1\right)$$

7.4.3 Using WKD-IBE for Signatures Efficiently

As we did for encryption (Section 7.3.6), we design JEDI to use WKD-IBE for signatures as efficiently as possible to reduce its resource overheads, with particular attention to low-power embedded devices.

7.4.3.1 Using WKD-IBE Rarely

In this section, we apply the techniques from Section 3.2.1 and Section 3.2.2 to JEDI's signatures, analogously to how we applied them in Section 7.3.6.1. As in Section 7.3.6.1, we must avoid computing a WKD-IBE signature for every message. A simple way to do this is to sample a digital signature keypair each hour, sign the verifying key with WKD-IBE at the beginning of the hour, and sign messages during the hour with the corresponding signing key.

Unfortunately, this may still be too expensive for low-power embedded devices because it requires a digital signature, which requires asymmetric-key cryptography, for *every* message. We can circumvent this by instead (1) choosing a *symmetric* key k every hour, (2) signing k at the start of each hour (using WKD-IBE for anonymity), and (3) using k in an *authenticated broadcast protocol* to authenticate messages sent during the hour. An authenticated broadcast protocol, like μTESLA [375], generates a MAC for each message using a key whose hash is the previous key; thus, the single signed key k allows the recipient to verify later messages, whose MACs are

generated with hash preimages of k. In general, this design requires stricter time synchronization than the one based on digital signatures, as the key used to generate the MAC depends on the time at which it is sent. However, for the sense-and-send use case typical of smart buildings, sensors anyway publish messages on a fixed schedule (e.g., one sample every x seconds), allowing the key to depend only on the message index. Thus, timely message delivery is the only requirement. Our scheme differs from μTESLA because the first key (end of the hash chain) is signed using WKD-IBE.

7.4.3.2 Precomputation with Adjustment

Additionally, we apply the technique from Section 3.2.4 to JEDI's anonymous signatures to develop a new signature algorithm for JEDI, analogously to how we used the technique to develop a new encryption algorithm in Section 7.3.6.2. Conceptually, **KeyDer**, which is used to produce signatures, can be understood as a two-step procedure: (1) produce a key of the correct form and structure (called **NonDelegableKeyDer**), and (2) re-randomize the key so that it can be safely delegated (called **ResampleKey**), as follows.
NonDelegableKeyDer(K,S): Parse K as (k_0,k_1,B), where $B=\{(i,b_i)\}$. Output:

$$\left(k_0 \cdot \prod_{\substack{(i,a_i)\in \text{fixed}(S)\\(i,b_i)\in B}} b_i^{a_i},\quad k_1,\quad \left\{(j,b_j)\right\}_{j\in\text{free}(S)}\right).$$

ResampleKey(K,S): Parse K as (k_0,k_1,B). Sample $t \xleftarrow{\$} \mathbb{Z}_p$ and output

$$\left(k_0 \cdot \left(g_3 \cdot \prod_{(i,a_i)\in \text{fixed}(S)} h_i^{a_i}\right)^t, g^t \cdot k_1, \left\{(j,h_j^t \cdot b_j)\right\}_{j\in\text{free}(S)}\right).$$

NonDelegableKeyDer and **ResampleKey** are the "two parts" of **KeyDer** in the sense that $\{\textbf{KeyDer}(K,S)\} = \{\textbf{ResampleKey}(\textbf{NonDelegableKeyDer}(K,S),S)\}$ where the distributions are over the sampled randomness.

We can take advantage of these functions to accelerate signing of messages. Note the similarity between **Sign** and **ResampleKey**. The setup we consider is that a principal has a key for some pattern R representing a URI prefix and time prefix. It will repeatedly sign messages with a pattern S representing at a fully-qualified URI and specific time, where R matches S. The next signature will be on pattern T which shares the same URI as S but corresponds to the next leaf in the time tree. The naïve algorithm is to call **QualifyKey** to obtain a key for S and then call **Sign**. The key idea behind the optimization is to instead call **NonDelegableKeyDer** to obtain a pseudo-key for S (which is not safe to delegate), and then create a signature for that. Observe that the resulting signature is distributed in exactly the same way whether the naïve or optimized method is used.

Now that we have described signing in terms of two calls, one to **NonDelegableKeyDer** and another to **Sign**, we describe how to apply precomputation with adjustment to each of these opera-

tions. **Sign** can be accelerated using the same precomputed value we used to accelerate encryption. We have already shown how to "adjust" this precomputed value from S to T.

SignPrepared(K, Q_S, m): Parse the key K as (k_0, k_1, B). Let S be the pattern corresponding to K; Q_S must be the precomputed value corresponding to S. Select $t \xleftarrow{\$} \mathbb{Z}_p^*$ and output:

$$\left(k_0 \cdot (h_s^m \cdot Q_S)^t, \quad g^t \cdot k_1\right)$$

Finally, we explain how the result of **NonDelegableKeyDer** can be adjusted from pattern S to pattern T. The procedure also requires the parent key (whose pattern we denote R), on which **NonDelegableKeyDer** was called to obtain the key corresponding to pattern S.

AdjustNonDelegable(P, C, S, T): Parse the parent key P as P as (p_0, p_1, B) where $B = \{(i, b_i)\}$. Parse the child key C as $C = (k_0, k_1, Z)$. S is the pattern corresponding to C, and T is the pattern that the resulting key will correspond to. Output:

$$\left(k_0 \cdot \prod_{\substack{(i,t_i)\in\text{fixed}(T)\\ i\in\text{free}(S)}} b_i^{t_i} \cdot \prod_{\substack{(i,s_i)\in\text{fixed}(S)\\ i\in\text{free}(T)}} b_i^{-s_i} \cdot \prod_{\substack{(i,s_i)\in\text{fixed}(S)\\ (i,t_i)\in\text{fixed}(T)}} b_i^{t_i - s_i}, \quad k_1, \quad \left\{(j, b_j)\right\}_{j\in\text{free}(T)}\right).$$

To sign a message each hour, JEDI maintains the result of **Precompute**, Q_S (as it does for encryption), and also the result of **NonDelegableKeyGen**, C, derived from its key. Then it adjusts both values, using **AdjustPrecomputed** and **AdjustNonDelegable**, when the pattern used to sign changes. To sign a message m, it computes **SignPrepared**(C, Q_S, m).

Additionally, we only compute the first two elements of the output of **NonDelegableKeyDer** and **AdjustNonDelegableKeyDer** when using it to produce signatures.

7.4.4 Security Guarantee

The integrity guarantees of the method in this section can be formalized using a game very similar to the one in Theorem 2, so we do not present it here for brevity. We do, however, formalize the anonymous aspect of WKD-IBE signatures:

Theorem 3. *For any well-formed keys k_1, k_2 corresponding to the same (URI, time) pair in the same resource hierarchy, and any message $m \in \mathbb{Z}_p^*$, the distribution of signatures over m produced using k_1 is information-theoretically indistinguishable from (i.e., equal to) the distribution of signatures over m produced using k_2.*

This implies that even a powerful adversary who observes the private keys held by all principals cannot distinguish signatures produced by different principals, for a fixed message and pattern. No computational assumptions are required.

Theorem 3 follows directly from the fact that the WKD-IBE construction used in JEDI is history-independent: each "signature" in WKD-IBE is the same as a private key generated with a special slot filled in with the message being signed. Therefore, signatures inherit the history-independence of keys, resulting in the property in Theorem 3. With the proposed improvement to

make signatures constant size (Section 7.4.2.2), the signature consists of just the first two terms of the resulting private key, but it remains history-independent nonetheless.

We now prove Theorem 3, using the same notation for signatures established in Section 7.3.1.2. The proof is very similar to the proof of Theorem 1.

Proof of Theorem 3. We will show that for any pattern S, key k corresponding to pattern S, and message m, it holds that

$$\{\mathbf{Sign}(k,m)\} = \left\{ \left(g_2^{\alpha} \cdot \left(g_3 \cdot h_s^m \cdot \prod_{(i,a_i)\in \text{fixed}(S)} h_i^{a_i} \right)^r,\ g^r \right) \right\}_{r \xleftarrow{\$} \mathbb{Z}_p}$$

Because the right-hand side of the above equation depends only on S and the public parameters (not the particular key k), this is sufficient to prove Theorem 3 (that any two keys corresponding to S produce the same signature distribution).

Observe that for a well-formed key k,

$$k = \left(g_2^{\alpha} \cdot \left(g_3 \cdot \prod_{(i,a_i)\in \text{fixed}(S)} h_i^{a_i} \right)^{r_0},\ g^{r_0},\ \left\{ (j, h_j^{r_0}) \right\}_{j\in \text{free}(S)} \right)$$

for some fixed $r_0 \in \mathbb{Z}_p$. Applying the formula for **Sign** in Section 7.3.1.2, the signature has the form

$$\left(g_2^{\alpha} \cdot \left(g_3 \cdot h_s^m \cdot \prod_{(i,a_i)\in \text{fixed}(S)} h_i^{a_i} \right)^{r_0+t},\ g^{r_0+t} \right)$$

for $t \xleftarrow{\$} \mathbb{Z}_p$. Because $r_0 + t$ is uniformly distributed in $\mathbb{Z}_p$, the output signature has the desired distribution (take $r = r_0 + t$). □

7.5 Implementation

We implemented JEDI as a library in the Go programming language. We expect that only a few applications will require the anonymous signature protocol in Section 7.4.2 or the immediate revocation protocol in our extended paper [297]; most applications can use signature chains (Section 7.4.1) for integrity and expiry for revocation (Section 7.3.8). Therefore, our implementation makes anonymous signatures optional and implements revocation separately. We expect JEDI's key delegation to be computed on relatively powerful devices, like laptops, smartphones, or Raspberry Pis; less powerful devices (e.g., right half of Figure 7.1) will primarily send and receive messages, rather than generate keys for delegation. Therefore, our focus for low-power platforms is on the "sense-and-send" use case [88, 157, 169] typical of indoor environmental sensing, where a device periodically publishes sensor readings to a URI. Whereas our Go library provides higher-level abstractions, we expect low-power devices to use JEDI's crypto library directly.

7.5.1 C/C++ Cryptography Library

As part of JEDI, we implemented a cryptography library optimized in assembly for three different architectures typical of IoT platforms (Figure 7.1). It implements WKD-IBE and JEDI's optimizations and modifications to it. The construction of WKD-IBE is based on a bilinear group in which the Bilinear Diffie-Hellman Exponent assumption holds. We originally planned to use Barreto-Naehrig elliptic curves [264, 114] to implement WKD-IBE. Unfortunately, a recent attack on Barreto-Naehrig curves [281] reduced their estimated security level from 128 bits to at most 100 bits [37]. Therefore, we use the recent BLS12-381 elliptic curve [81].

State-of-the-art cryptography libraries implement BLS12-381, but none of them, to our knowledge, optimize for microarchitectures typical of low-power embedded platforms. To improve energy consumption, we implemented BLS12-381 in C/C++, profiled our implementation, and rewrote performance-critical routines in assembly. We focus on ARM Cortex-M, an IoT-focused family of 32-bit microprocessors typical of modern low-power embedded sensor platforms [210, 96, 239]. Cortex-M processors have been used in billions of devices, including commercial IoT offerings such as Fitbit and Nest Protect. Our assembly targets Cortex-M0+, which is among the least powerful of processors in the Cortex-M series, and of those used in IoT devices (farthest to the right in Figure 7.1). By demonstrating the practicality of JEDI on Cortex-M0+, we establish that JEDI is viable across the spectrum of IoT devices (Figure 7.1).

The main challenge in targeting Cortex-M0+ is that the 32-bit multiply instruction provides only the lower 32 bits of the product. Even on more powerful microarchitectures without this limitation (e.g., Intel Core i7), most CPU time ($\geq 80\%$) is spent on multiply-intensive operations (e.g., BigInt multiplication and Montgomery reduction), so the lack of such an instruction was a performance bottleneck. As a workaround, our assembly code emulates multiply-accumulate with carry in 23 instructions. Cortex-M3 and Cortex-M4, which are more commonly used than Cortex-M0+, have instructions for 32-bit multiply-accumulate which produce the entire 64-bit result; we expect JEDI to be more efficient on those processors. We also wrote assembly to optimize BLS12-381 for x86-64 and ARM64, representative of server/laptop and smartphone/Raspberry Pi, respectively (first two tiers in Figure 7.1). We were able to improve performance by minimizing data transfer between registers and the cache in multiply-intensive operations. Thus, our Go library, which runs on these non-low-power platforms, also benefits from low-level assembly optimizations. Writing assembly code by hand to better support cryptography, as we did in JEDI, can be seen as an application of the technique in Section 3.1.1.

7.5.2 Application of JEDI to bw2

We used our JEDI library to implement end-to-end encryption in bw2, a syndication and authorization system for IoT. bw2's syndication model is based on publish-subscribe, explained in Section 7.1. Here we discuss bw2's authorization model. Access to resources is granted via certificate chains from the authority of a resource hierarchy to a principal. Individual certificates are called Declarations of Trust (DOTs). bw2 maintains a publicly accessible registry of DOTs, implemented using blockchain smart contracts, so that principals can find the DOTs they need to form DOT

chains. A *trusted* router enforces permissions granted by DOTs. Principals must present DOT chains when publishing/subscribing to resources, and the router verifies them. Note that a compromised router can read messages.

We use JEDI to enforce bw2's authorization semantics with end-to-end encryption. DOTs granting permission to subscribe now contain WKD-IBE keys to decrypt messages. By default, DOTs granting permission to publish to a URI remain unchanged, and are used as in Section 7.4.1. WKD-IBE keys may also be included in DOTs granting publish permission, for anonymous signatures (Section 7.4.2). Using our library for JEDI, we implemented a wrapper around the bw2 client library. It transparently encrypts and decrypts messages using WKD-IBE, and includes WKD-IBE parameters and keys in DOTs and principals, as needed for JEDI. bw2 signs each message with a digital signature (first alternative in Section 7.4.3).

The bw2-specific wrapper is less than 900 lines of Go code. Our implementation required no changes to bw2's client library, router, blockchain, or core—it is a separate module. Importantly, it provides the same API as the standard bw2 client library. Thus, it can be used as a drop-in replacement for the standard bw2 client library, to easily add end-to-end encryption to existing bw2 applications with minimal changes.

7.6 Evaluation

We first compare JEDI's underlying encryption scheme, WKD-IBE, to alternatives, in order to evaluate the benefits of choosing WKD-IBE according to the technique in Section 3.2.3. We then evaluate JEDI via microbenchmarks, determine its power consumption on a low-power sensor, measure the overhead of applying it to bw2, and compare it to other systems.

7.6.1 Building Block Comparison: HIBE, WKD-IBE, and KP-ABE

We first explore two alternative encryption schemes that we could have used for JEDI: HIBE and KP-ABE. Then we compare the costs of these two alternatives to WKD-IBE, the encryption scheme that JEDI uses. Finally, we discuss other HIBE variants. Our purpose in doing this analysis is to demonstrate the benefits of choosing WKD-IBE according to the technique in Section 3.2.3.

7.6.1.1 Hierarchical Identity-Based Encryption (HIBE)

Given that JEDI represents URIs and time as hierarchies, Hierarchical Identity-Based Encryption (HIBE) [63] may seem like a natural building block to use. We can encode a URI in an ID for HIBE, just as we did for WKD-IBE. For example, the URI prefix `a/b/*` can be encoded into an ID as `("a", "b")`. This preserves the crucial property that the private key for a URI prefix can be used to generate the private key for any URI with that prefix. The same thing works for expiry: for example, the timestamp June 08, 2017 at 6 AM could be encoded into an ID as `("2017", "June", "08", "06")`.

However, HIBE cannot *simultaneously* support a URI hierarchy and an expiry hierarchy. A simple approach would be to concatenate the IDs. For example, the key for the URI prefix `a/b/*`

and time prefix `2018/Jun/*` would have the ID `("2018", "Jun", "a", "b")`. However, this idea is flawed: only the URI can be extended, not the expiry time. The same problem applies to the URI, if we put the URI before the time in the ID. Another possible approach is to interleave the resource hierarchy and time hierarchy, using metadata to distinguish the elements. In this setup, each (resource, time) pair corresponds to multiple IDs in the HIBE system—all possible interleavings of the URI the Expiry IDs. However, for a URI of length m and a time of length n, there are exponentially many IDs, $\frac{(m+n)!}{m!\cdot n!}$, and each message sent with that URI and time must be encrypted under all of those IDs. Therefore, this approach is infeasible.

Another strawman is to use two HIBE systems, one for URIs and one for expiry. Each message is encrypted twice, using the URI ID in the first system and again using the time ID in the second system. During delegation, each principal is provided with a key from the first system for the URI, and a set of keys from the second system for the time range. The problem is that this approach is not collusion-resistant: a principal who is given two delegations, one for the correct URI that has expired, and one for the wrong URI that has not expired, can decrypt messages by combining keys from different delegations.

7.6.1.2 Key-Policy Attribute-Based Encryption (KP-ABE)

In Key-Policy Attribute-Based Encryption (KP-ABE), a message is encrypted with a set of attributes. An attribute set is like a string of bits; each attribute is either present in the set (1) or not present (0). Private keys are generated with an *access tree*, which can be thought of as a circuit. A private key can decrypt a message if its access tree, evaluated on the bits representing the attribute set of the message, evaluates to 1.

We are interested in KP-ABE with two properties:

1. **Delegable.** Given the private key for an access tree, one can generate a private key for a more restrictive access key, and delegate it to another principal.

2. **Large Universe.** The space of attributes $\mathcal{A}$ is exponentially large in the security parameter κ. This is similar to Identity-Based Encryption (IBE) [65], as any string of bytes can be hashed to an attribute.

The GPSW construction [203] of KP-ABE, based on bilinear groups, satisfies these properties. In fact, KP-ABE with these two properties subsumes WKD-IBE. A pattern T in WKD-IBE can be converted to an attribute set in Delegable Large Universe KP-ABE by hashing each non-$\perp$ component of T, concatenated with its index, to an attribute in KP-ABE. Private keys in WKD-IBE can be expressed as an access tree consisting of a single many-input AND gate.[3]

[3]Ciphertext-Policy ABE (CP-ABE) is not suitable for this construction. This is because attributes cannot be added to secret keys during delegation, as per the security guarantees of CP-ABE.

Operation	Scheme	Pairings	Exponentiations
Encrypt	HIBE	0	$3+r$
Encrypt	WKD-IBE	0	$3+r$
Encrypt	KP-ABE	0	$2+r\cdot(\ell+3)$
Decrypt	HIBE	2	$\leq r$
Decrypt	WKD-IBE	2	$\leq r$
Decrypt	KP-ABE	$r+1$	$2r$
KeyDer[1]	HIBE	0	$\ell+2$
KeyDer[1]	WKD-IBE	0	$\ell+2$
KeyDer[1]	KP-ABE	0	$r\cdot(\ell+5)$
KeyDer[2]	HIBE	0	$(r-n)+\ell+2$
KeyDer[2]	WKD-IBE	0	$(r-n)+\ell+2$
KeyDer[2]	KP-ABE	0	$2n+r\cdot(\ell+5)$

Table 7.1: Performance comparison of HIBE, WKD-IBE, and KP-ABE in terms of pairings and exponentiations. We omit operations that can be precomputed once for all IDs (attribute sets) in the HIBE/WKD-IBE/KP-ABE system. **KeyDer[1]** indicates deriving the new key from the master key, and **KeyDer[2]** indicates the other case.

7.6.1.3 Performance Comparison

We compare the performance of KP-ABE, WKD-IBE, and HIBE in terms of the number of exponentiations and pairings, the most expensive operations in the elliptic curves. This is shown in Table 7.1. ℓ is the total number of attributes that can be used for a single message (the implicit argument to **Setup**). For Encrypt, Decrypt and KeyDer[1], r is the number of attributes of the key or ciphertext. For KeyDer[2], n is the number of attributes of the starting key, and r is the number of attributes of the ending key. This shows that WKD-IBE's performance is theoretically better than KP-ABE's performance. Furthermore, WKD-IBE is just efficient as HIBE, even though WKD-IBE is more expressive than HIBE. As discussed in Section 7.6.1.1, HIBE is not expressive enough to efficiently instantiate JEDI.

7.6.1.4 Size Comparison

We list the size of ciphertexts and private keys in Table 7.2: r is the number of attributes in the ciphertext or private key, and ℓ is the maximum number of slots or attributes used to encrypt a message. Note that ciphertexts in WKD-IBE are constant size, whereas ciphertexts in KP-ABE are linear.

Object	Scheme	$\mathbb{G}_1$	$\mathbb{G}_2$	$\mathbb{G}_T$
Ciphertext	HIBE	1	1	1
Ciphertext	WKD-IBE	1	1	1
Ciphertext	KP-ABE	r	1	1
Private Key	HIBE	$\ell - r + 1$	1	0
Private Key	WKD-IBE	$\ell - r + 1$	1	0
Private Key	KP-ABE	r	r	0

Table 7.2: Size comparison of HIBE, WKD-IBE, and KP-ABE in terms of number of group elements. For elliptic curves that we used, elements of $\mathbb{G}_1$ are 48 B each, elements of $\mathbb{G}_2$ are 96 B each, and elements of $\mathbb{G}_T$ are 576 B each.

Operation	Laptop	Raspberry Pi	Sensor
$\mathbb{G}_1$ Multiplication (Chosen Scalar)	109 µs	1.33 ms	509 ms
$\mathbb{G}_2$ Multiplication (Chosen Scalar)	343 µs	3.86 ms	1.44 s
$\mathbb{G}_T$ Multiplication (Random Scalar)	504 µs	5.47 ms	1.90 s
$\mathbb{G}_T$ Multiplication (Chosen Scalar)	507 µs	5.48 ms	2.81 s
Pairing	1.29 ms	14.0 ms	4.99 s

Table 7.3: Latency of JEDI's implementation of BLS12-381.

7.6.1.5 Variants of HIBE Other Than WKD-IBE

Existing work [497] has proposed extending HIBE to MHIBE, which supports ID-based encryption for multiple concurrent hierarchies. We could use MHIBE in JEDI to combine the URI hierarchy with the expiry hierarchy. However, the proposed MHIBE schemes are significantly less performant than WKD-IBE: for two hierarchies, they have quadratically-sized private keys and ciphertexts. Size and performance degrade exponentially in the number of hierarchies. Furthermore, a formal treatment of MHIBE is not provided.

Another extension is forward secure HIBE [497, 63], or fs-HIBE for short. The BBG construction of fs-HIBE [63] has linear size and performance. We considered using its mechanism for forward security *in reverse*, to achieve expiry with HIBE. However, the fs-HIBE construction has linear-size ciphertexts and linear-time decryption in the depth of the time hierarchy, whereas WKD-IBE has constant-size ciphertexts and constant-time decryption. In the context of a real system, this is important: **Encrypt** and **Decrypt** are used in the critical path, so encryption time, decryption time, and ciphertext size must be as small as possible. In contrast, **Delegate** is only used occasionally, so the size of private keys is less important.

Most importantly, WKD-IBE is a more powerful primitive than either MHIBE or fs-HIBE. In particular, WKD-IBE supports the + wildcard for URIs and timestamps (Section 7.3.10.1), which MHIBE and fs-HIBE do not.

	Laptop	Raspberry Pi
Encrypt	3.08 ms	37.3 ms
Decrypt	3.61 ms	43.9 ms
KeyDer	4.77 ms	58.5 ms
Sign	4.80 ms	61.2 ms
Verify	4.78 ms	56.3 ms

Figure 7.6: Latency of **Encrypt**, **Decrypt**, **KeyDer**, **Sign**, and **Verify** with 20 attributes.

7.6.2 Microbenchmarks

Benchmarks labeled "Laptop" were produced on a Lenovo T470p laptop with an Intel Core i7-7820HQ CPU @ 2.90 GHz. Benchmarks labeled "Raspberry Pi" were produced on a Raspberry Pi 3 Model B+ with an ARM Cortex-A53 @ 1.4 GHz. Benchmarks labeled "Sensor" were produced on a commercially available ultra low-power environmental sensor platform called "Hamilton" with an ARM Cortex-M0+ @ 48 MHz. We describe Hamilton in more detail in Section 7.6.4.

7.6.2.1 Performance of BLS12-381 in JEDI

Table 7.3 compares the performance of JEDI's BLS12-381 implementation on the three platforms, with our assembly optimizations. As expected from Figure 7.1, the Raspberry Pi performance is an order of magnitude slower than Laptop performance, and performance on the Hamilton sensor is an additional two-to-three orders of magnitude slower.

7.6.2.2 Performance of WKD-IBE in JEDI

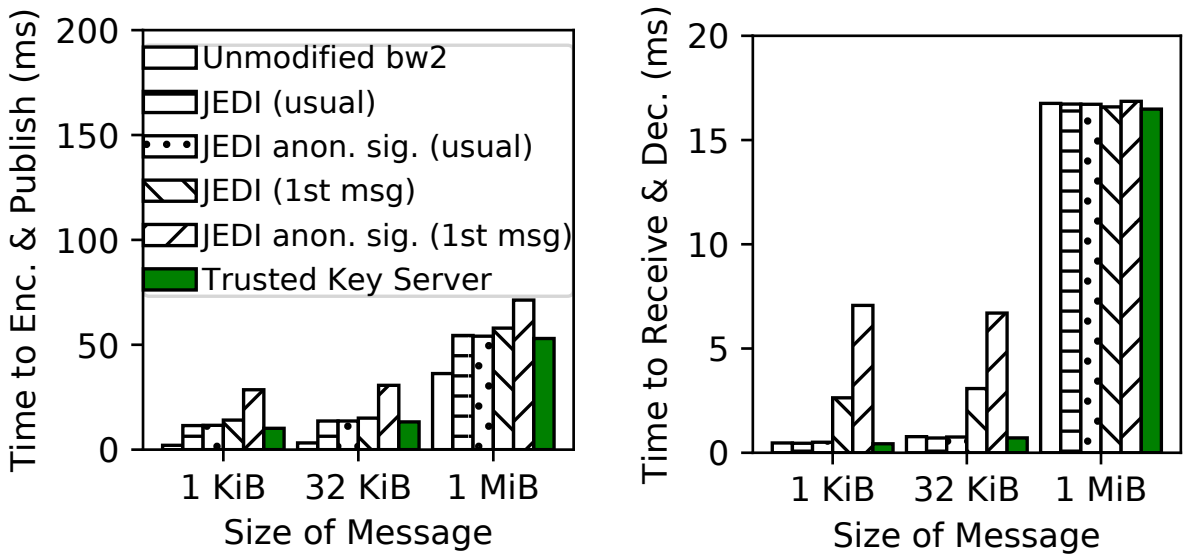
Figure 7.6 depicts the performance of JEDI's cryptography primitives. Figure 7.6 does not include the sensor platform; Section 7.6.4 thoroughly treats performance of JEDI on low-power sensors.

In Figure 7.6, we used a pattern of length 20 for all operations, which would correspond to, e.g., a URI of length 14 and an Expiry hierarchy of depth 6. To measure decryption and signing time, we measure the time to decrypt the ciphertext or sign the message, plus the time to generate a decryption key for that pattern or ID. For example, if one receives a message on `a/b/c/d/e/f`, but has the key for `a/*`, he must generate the key for `a/b/c/d/e/f` to decrypt it.

Figure 7.6 demonstrates that the JEDI encrypts and signs messages and generates qualified keys for delegation at practical speeds. On a laptop, all WKD-IBE operations take less than 10 ms with up to 20 attributes. On a Raspberry Pi, they are 10× slower (as expected), but still run at interactive speeds.

7.6.3 Performance of JEDI in bw2

In bw2 (Section 7.5.2), the two critical-path operations are publishing a message to a URI, and receiving a message as part of a subscription. We measure the overhead of JEDI for these opera-



(a) Encrypt/publish message.

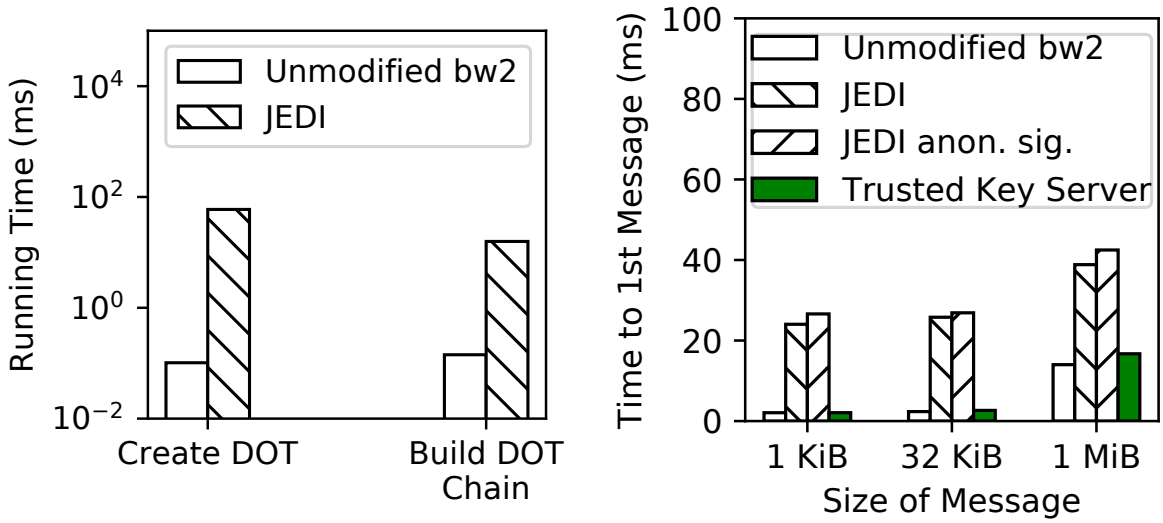
(b) Receive/decrypt message.

Figure 7.7: Critical-path operations in bw2, with/without JEDI.

tions because they are core to bw2's functionality and would be used by any messaging application built on bw2. Our methodology is to perform each operation repeatedly in a loop, to measure the sustained performance (operations/second), and report the average time per operation (inverse). To minimize the effect of the network, the router was on the same link as the client, and the link capacity was 1 Gbit/s. In our experiments, we used a URI of length 6 and an Expiry tree of depth 6. We also include measurements from a strawman system with pre-shared AES keys—this represents the critical-path overhead of an approach based on the Trusted Key Server discussed in Section 7.2. Our results are in Figure 7.7.

We implement the optimizations in Section 7.3.6.1, so only symmetric key encryption/decryption must be performed in the common case (labeled "usual" in the diagram). However, the symmetric keys will *not* be cached for the first message sent every hour, when the WKD-IBE pattern changes. A WKD-IBE operation must be performed in this case (labeled "1st message" in the diagram). For large messages, the cost of symmetric key encryption dominates. JEDI has a particularly small overhead for 1 MiB messages in Figure 7.7b, perhaps because 1 MiB messages take several milliseconds to transmit over the network, allowing the client to decrypt a message while the router is sending the next message.

We also consider creating DOTs and initiating subscriptions, which are not in the critical path of bw2. These results are in Figure 7.8 (note the log scale in Figure 7.8a). Creating DOTs is slower with JEDI, because WKD-IBE keys are generated and included in the DOT. Initiating a subscription in bw2 requires forming a DOT chain; in JEDI, one must also derive a private key from the DOT chain. Figure 7.8a shows the time to form a short one-hop DOT chain, and in the



(a) Create DOT, build chain.

(b) Time to query/subscribe.

Figure 7.8: Occasional bw2 operations, with and without JEDI.

case of JEDI, includes the time to derive the private key. For JEDI's encryption (Section 7.3), these additional costs are incurred only by DOTs that grant permission to subscribe. With anonymous signatures, DOTs granting permission to publish incur this overhead as well, as WKD-IBE keys must be included. Figure 7.8b puts this in context by measuring the end-to-end latency from initiating a subscription to receiving the first message (measured using bw2's "query" functionality).

For a DOT to be usable, it must be inserted into bw2's registry. This requires a blockchain transaction (not included in Figure 7.8). An important consideration in this regard is *size*. In the unmodified bw2 system, a DOT that grants permission on `a/b/c/d/e/f` is 198 bytes. With JEDI, each DOT also contains multiple WKD-IBE keys, according to the time range. In the "worst case," where the start time of a DOT is Jan 01 at 01:00, and the end time is Dec 31 at 22:59, a total of 45 keys are needed. Each key is approximately 1 KiB (Table 7.2), so the size of this DOT is approximately 45 KiB.

Because bw2's registry of DOTs is implemented using blockchain smart contracts, the bandwidth for inserting DOTs is limited. Using JEDI would increase the size of DOTs as above, resulting in an approximately 100–400× decrease in aggregate bandwidth for creating DOTs. However, this can be mitigated by changing bw2 to not store DOTs directly in the blockchain. DOTs can be stored in untrusted storage, with only their hashes stored in the blockchain-based registry. Such a solution could be based on Swarm [454] or Filecoin [171].

Operation	Time	Average Current
Sleep (Idle)	N/A	0.0063 mA
WKD-IBE Encrypt	6.50 s	10.2 mA
WKD-IBE Encrypt and Sign	9.89 s	10.2 mA

Table 7.4: CPU and power costs on the Hamilton platform.

7.6.4 Feasibility on Ultra Low-Power Devices

We use a commercially available sensor platform called "Hamilton" [210, 15] built around the Atmel SAMR21 system-on-chip (SoC). The SAMR21 costs approximately $2.25 per unit [143] and integrates a low-power microcontroller and radio. The sensor platform we used in this study costs $18 to manufacture [274]. For battery lifetime calculations, we assume that the platform is powered using a CR123A Lithium battery that provides 1400 mAh at 3.0 V (252 J of energy). Such a battery costs $1. The SAMR21 is heavily constrained: it has only a 48 MHz CPU frequency based on the ARM Cortex-M0+ microarchitecture, and a total of only 32 KiB of data memory (RAM). Our goal is to validate that JEDI is practical for an ultra low-power sensor platform like Hamilton, in the context of a "sense-and-send" application in a smart building. Since most of the platform's cost ($18) comes from the on-board transducers and assembly, rather than the SAMR21 SoC, *using an even more resource-constrained SoC would not significantly decrease the platform's cost.* An analogous argument applies to energy consumption, as the transducers account for more than half of Hamilton's idle current [274].

Hamilton/SAMR21 is on the lower end of platforms typically used for sense-and-send applications in buildings. Some older studies [169, 314] use even more constrained hardware like the TelosB; this is because those studies were constrained by hardware available at the time. Modern 32-bit SoCs, like the SAMR21, offer substantially better performance at a similar price/power point to those older platforms [274].

7.6.4.1 CPU Usage

Table 7.4 shows the time for encryption and anonymous signing in JEDI on Hamilton. The results use the optimizations discussed in Section 7.3.6 and Section 7.4.3, and include the time to "adjust" precomputed state. They indicate that symmetric keys can be encrypted and anonymously signed in less than 10 seconds. This is feasible given that encryption and anonymous signing occur rarely, once an hour, and need not be produced at interactive speeds in the normal "sense-and-send" use case.

7.6.4.2 Power Consumption

To calculate the impact on battery lifetime, we consider a "sense-and-send" application, in which the Hamilton device obtains readings from its sensors at regular intervals, and immediately sends the readings encrypted over the wireless network. We measured the average current consumed for

	AES Only	JEDI (Encrypt)	JEDI (Encrypt and Sign)
10 s	32 µA / 5.1 y	50 µA / 3.2 y	60 µA / 2.6 y
20 s	20 µA / 8.1 y	38 µA / 4.2 y	48 µA / 3.3 y
30 s	15 µA / 10 y	34 µA / 4.7 y	44 µA / 3.6 y

Table 7.5: Average current and expected battery life (for 1400 mAh battery) for sense-and-send, with varying sample interval.

varying sample intervals, when each message is encrypted with AES-CCM, without using JEDI ("AES Only" in Table 7.5). We estimate JEDI's average current based on the current, duration, and frequency (once per hour, for these estimates) of JEDI operations, and add it to the average current of the "AES Only" setup. Our estimates assume that the µTESLA-based technique in Section 7.4.3 is used to avoid attaching a digital signature to each message. We divide the battery's energy capacity by the result to compute lifetime. As shown in Table 7.5, JEDI decreases battery life by about 40-60%. Battery life is several years even with JEDI, acceptable for IoT sensor platforms.

JEDI's overhead depends primarily on the granularity of expiry times (one hour, for these estimates), *not* the sample interval. To improve power consumption, one could use a time tree with larger leaves, allowing principals to perform WKD-IBE encryptions and anonymous signatures less often. This would, of course, make expiry times coarser.

7.6.4.3 Memory Budget

Performing WKD-IBE operations requires only 6.5 KiB of data memory, which fits comfortably within the 32 KiB of data memory (RAM) available on the SAMR21. The code space required for our implementation of WKD-IBE and BLS12-381 is about 74 KiB, which fits comfortably in the 256 KiB of code memory (ROM) provided by the SAMR21.

A related question is whether storing a hash chain in memory (as required for authenticated broadcast, Section 7.4.3) is practical. If we use a granularity of 1 minute for authenticated broadcast, the length of the hash chain is 60. At the start of an hour, one computes the entire chain, storing 10 hashes equally spaced along the chain, each separated by 5 hashes. As one progresses along the hash chain, one re-computes each set of 5 hashes one additional time. This requires storage for only 15 hashes ($<$ 4 KiB memory) and computation of only 105 hashes *per hour*, which is practical. One could possibly optimize performance further using *hierarchical hash chains* [225].

7.6.4.4 Impact of Techniques from Chapter 3

JEDI's cryptographic optimizations (Section 7.3.6.2, Section 7.4.2.2, Section 7.4.3), which apply the technique from Section 3.2.4 to use WKD-IBE in a non-black-box manner, provide a 2–3$\times$ performance improvement. Our assembly optimizations (Section 7.5), which can be seen as an application of the technique from Section 3.1.1, provide an additional 4–5$\times$ improvement. Hybrid encryption and key reuse (Section 7.3.6.1), which apply the techniques from Section 3.2.1

and Section 3.2.2 are also crucial because they allow JEDI to use WKD-IBE *rarely*, with a frequency tied to the granularity of expiry times rather than to the frequency with which messages are sent. And of course, it is crucial for JEDI to use WKD-IBE instead of ABE (technique from Section 3.2.3), which is much more expensive (likely an order of magnitude more expensive), as discussed in Section 7.6.1. All of these techniques contribute to making JEDI practical on ultra low-power embedded sensing devices.

7.6.5 Comparison to Other Systems

Crypto Scheme / System	Avoids Central Trust?	Expressivity	Performance
Trusted Key Server (Section 7.2)	– No	+ Supports arbitrary policies (beyond hierarchies) – No delegation	+ ≈ 10 µs to encrypt 1 KiB message (same as JEDI in common case, faster for first message after key rotation) – Trusted party generates one key *per resource*
PRE (Lattice-Based), as used in PICADOR [75]	– No	+ Supports arbitrary policies (beyond hierarchies) – No delegation	+ ≈ 5 ms encrypt, ≈ 3 ms decrypt (similar to JEDI: 3–4 ms) – Trusted party must generate one key per sender-receiver pair
PRE (Pairing-Based), as used in Pilatus [419]	+ Yes	– Delegation is single-hop – Delegation is coarse (all-or-nothing) + Can compute aggregates on encrypted data	+ 0.6 ms encrypt, 1.3 ms re-encrypt, 0.5 ms decrypt (faster than JEDI: 3–4 ms) + Practical on constrained IoT device with crypto accelerator
CP-ABE [51]	+ Yes	+ Good fit for RBAC policies – Cannot support JEDI's hierarchy abstraction with delegation	+ Only symmetric crypto in common case – 14 ms encrypt for first time after key rotation (4–5× slower than JEDI: 3 ms)

KP-ABE, as used in Sieve [468]	+ Yes	+ Succinct delegation based on attributes – Delegation is single-hop	+ Only symmetric crypto in common case – 25 ms encrypt for first time after key rotation (8–9× slower than JEDI: 3 ms)
Delegable Large Univ. KP-ABE [203] (used in Alternative JEDI Design)	+ Yes	+ Generalizes beyond hierarchies and supports multi-hop delegation (subsumes JEDI)	+ Only symmetric crypto in common case – 60 ms encrypt for first time after key rotation (20× slower than JEDI: 3 ms) – Impractical for low-power sense-and-send
Our work: WKD-IBE [2] with Optimizations, as used in JEDI	+ Yes	+ Delegation is multi-hop + Succinct delegation of *subtrees* of resources (or more complex sets, Section 7.3.10) + Non-interactive expiry	+ After key rotation (e.g., once per hour), 3 ms encrypt, 4 ms decrypt (Figure 7.6) + Only symmetric crypto in common case + Practical for ultra low-power "sense-and-send" *without crypto accelerator*

Table 7.6: Comparison of JEDI with other crypto-based IoT/cloud systems.

Table 7.6 compares JEDI to other systems and cryptographic approaches, particularly those geared toward IoT, in regard to security, expressivity and performance. We treat these existing systems as they would be used in a messaging system for smart buildings (Section 7.1). Table 7.6 contains quantitative comparisons to the cryptography used by these systems; for those schemes based on bilinear groups, we re-implemented them using our JEDI crypto library (Section 7.5.1) for a fair comparison.

7.6.5.1 Security

The owner of a resource is considered *trusted* for that resource, in the sense that an adversary who compromises a principal can read all of that principal's resources. In Table 7.6, we focus on whether a single component is trusted for *all* resources in the system. Note that, although Trusted Key Server (Section 7.2) and PICADOR [75] encrypt data in flight, granting or revoking access to a principal requires participation of an *online trusted party* to generate new keys.

7.6.5.2 Expressivity

PRE-based approaches, which associate public keys with users and support delegation via proxy re-encryption, are coarse-grained—a re-encryption key allows *all* of a user's data to be re-encrypted. While PICADOR [75], a PRE-based system, allows more fine-grained semantics, but does not enforce them cryptographically. In short, the underlying encryption schemes are not expressive enough to capture the semantics of fine-grained access control. While certain ABE schemes are expressive enough to support delegation, ABE-based approaches typically do not support delegation beyond a single hop, whereas JEDI achieves multi-hop delegation. An advantage of ABE-based schemes is that attributes/policies attached to keys can describe more complex sets of resources than JEDI, because ABE is more expressive than WKD-IBE. In JEDI, we use WKD-IBE instead of ABE in order to leverage the expressivity-efficiency trade-off (Section 2.2.2)—WKD-IBE is less expressive than ABE, but it is more efficient than ABE. In the context of JEDI's intended use case, namely smart cities, this decision is well-grounded; existing syndication systems for smart cities, which do not encrypt data and are unconstrained by the expressiveness of crypto schemes, choose a hierarchical rather than attribute-based representation (Section 7.1), providing evidence that our WKD-IBE-based construction is expressive enough to capture IoT use cases.

7.6.5.3 Performance

The Trusted Key Server (Section 7.2) is the most naïve approach, requiring an online trusted party to enforce all policy. Even so, JEDI's performance in the common case is the same as the Trusted Key Server (Figure 7.7), because of JEDI's hybrid encryption—JEDI invokes WKD-IBE *rarely*. Even when JEDI invokes WKD-IBE, its performance is not significantly worse than PRE-based approaches. An alternative design for JEDI uses the GPSW KP-ABE construction instead of WKD-IBE, but it is significantly more expensive. Based Table 7.5, the power cost of a WKD-IBE operation *even when only invoked once per hour* contributes significantly to the overall energy consumption on the low-power IoT device; using KP-ABE instead of WKD-IBE would increase this power consumption by an order of magnitude, reducing battery life significantly.

7.6.5.4 Summary

In summary, existing systems fall into one of three categories. (1) The Trusted Key Server allows access to resources to be managed by arbitrary policies, but relies on a *central trusted party* who must be online whenever a user is granted access or is revoked. (2) PRE-based approaches, which permit sharing via re-encryption, cannot cryptographically enforce fine-grained policies or support multi-hop delegation. (3) ABE-based approaches, if carefully designed, can achieve the same expressivity as JEDI or even greater expressivity than JEDI, but are substantially less performant and are not suitable for low-power embedded devices.

7.7 Related Work

SiRiUS [190] and Plutus [261] are encrypted filesystems based on traditional public-key cryptography, but they do not support delegable and qualifiable keys like JEDI. Akl et al. [4] and further work [129, 130] propose using key assignment schemes for access control in a hierarchy. A line of work [456, 228, 22, 21] builds on this idea to support both hierarchical structure and temporal access. Key assignment approaches, however, require the full hierarchy to be known at setup time, which is not flexible in the IoT setting. JEDI does not require this, allowing different subtrees of the hierarchy to be managed separately (Section 7.1.2, "Delegation").

Tariq et al. [446] use Identity-Based Encryption (IBE) [65] to achieve end-to-end encryption in publish-subscribe systems, without the router's participation in the protocol. However, their approach does not support hierarchical resources. Further, encryption and private keys are on a credential-basis, so each message is encrypted multiple times according to the credentials of the recipients. Wu et al. [490] use a prefix encryption scheme based on IBE for mutual authentication in IoT. Their prefix encryption scheme is different from JEDI, in that users with keys for identity `a/b/c` can decrypt messages encrypted with prefix identity `a`, `a/b` and `a/b/c`, but not identities like `a/b/c/d`.

Since the original proposal of Hierarchical Identity-Based Encryption (HIBE) [186], there have been multiple HIBE constructions [62, 63, 183] and variants of HIBE [497, 2]. Although seemingly a good match for resource hierarchies, HIBE cannot be used as a black box to efficiently instantiate JEDI. We considered alternative designs of JEDI based on existing variants of HIBE, but as we elaborate in Section 7.6.1.1, each resulting design is either less expressive or significantly more expensive than JEDI.

A line of work [501, 468] uses Attribute-Based Encryption (ABE) [203, 51] to delegate permission. For example, Yu et al. [501] and Sieve [468] use Key-Policy ABE (KP-ABE) [203] to control which principals have access to encrypted data in the cloud. Some of these approaches also provide a means to revoke users, leveraging proxy re-encryption to safely perform re-encryption in the cloud. Our work additionally supports hierarchically-organized resources and decentralized delegation of keys, which [501] and [468] do not address. As discussed in Section 7.6.1 and Section 7.6.5, WKD-IBE is substantially more efficient than KP-ABE and provides enough functionality for JEDI. WKD-IBE could be a lightweight alternative to KP-ABE for some applications. Other approaches prefer Ciphertext-Policy ABE (CP-ABE) [51]. Existing work [470, 471] combines HIBE with CP-ABE to produce Hierarchical ABE (HABE), a solution for sharing data on untrusted cloud servers. The "hierarchical" nature of HABE, however, corresponds to the hierarchical organization of domain managers in an enterprise, not a hierarchical organization of *resources* as in our work.

NuCypher KMS [159] allows a user to store data in the cloud encrypted under her public key, and share it with another user using Proxy Re-Encryption (PRE) [58]. While NuCypher assumes limited collusion among cloud servers and recipients (e.g., m of n secret sharing) to achieve properties such as expiry, JEDI enforces expiry via cryptography, and therefore remains secure against *any* amount of collusion. Furthermore, NuCypher's solution for resource hierarchies requires a keypair for each node in the hierarchy, meaning that the creation of resources is centralized. Fi-

nally, keys in NuCypher are not qualifiable. Given a key for `a/*`, one cannot generate a key for `a/b/*` to give to another principal.

PICADOR [75], a publish-subscribe system with end-to-end encryption, uses a lattice-based PRE scheme. However, PICADOR requires a central Policy Authority to specify access control, by creating a re-encryption key for every permitted pair of publisher and subscriber. In contrast, JEDI's access control is decentralized.

Broadcast encryption (BE) [354, 145, 66, 69, 313, 71, 72] is a mechanism to achieve revocation, by encrypting messages such that they are only decryptable by a specific set of users. However, these existing schemes do not support key qualification and delegation, and therefore, cannot be used in JEDI directly. Another line of work builds revocation directly into the underlying cryptography primitive, achieving Revocable IBE [61, 318, 417, 478], Revocable HIBE [415, 416, 323] and Revocable KP-ABE [24]. These papers use a notion of revocation in which URIs are revoked. In contrast, JEDI supports revocation at the level of keys. If multiple principals have access to a URI, and one of their keys is revoked, then the other principal can still use its key to access the resource. Some systems [159, 43] rely on the participation of servers or routers to achieve revocation.

Secure Reliable Multicast [331, 333] also uses a many-to-many communication model, and ensures correct data transfer in the presence of malicious routers. JEDI, as a protocol to *encrypt* messages, is complementary to those systems.

JEDI is complementary to authorization services for IoT, such as bw2 [16], Vanadium [444], WAVE [17], and AoT [355], which focus on expressing authorization policies and enabling principals to prove they are authorized, rather than on encrypting data. Droplet [418] provides encryption for IoT, but does not support delegation beyond one hop and does not provide hierarchical resources.

An authorization service that provides secure in-band permission exchange, like WAVE [17], can be used for key distribution in JEDI. JEDI can craft keys with various permissions, while WAVE can distribute them without a centralized party by including them in its attestations. As we describe in Section 9.1.2, we integrated JEDI into WAVE and used the resulting system to collect sensor data from buildings in California.

7.8 Conclusion

In this chapter, we presented JEDI, a protocol for end-to-end encryption for IoT. JEDI provides *many-to-many* encrypted communication on complex resource hierarchies, supports decentralized key delegation, and decouples senders from receivers. It provides expiry for access to resources, reconciles anonymity and authorization via anonymous signatures, and, as described in our extended paper [297], allows revocation via tree-based broadcast encryption. Importantly, it can provide similar semantics as *unencrypted* IoT systems that are not constrained by cryptography. While policy-based encryption, like ABE, enables these semantics, a naïve solution that invokes ABE for each message would have been far from practical for resource-constrained IoT devices. The techniques from Section 3.2 were crucial to making JEDI practical for such devices. They led

us to (1) use WKD-IBE, a cheaper encryption scheme than ABE, (2) use WKD-IBE rarely and off of the critical path of individual messages, (3) tie the frequency of WKD-IBE operations to the granularity of expiry times to make JEDI's costs flexible, and (4) develop a specialized interface to WKD-IBE to produce WKD-IBE ciphertexts and signatures for JEDI more efficiently. Additionally, we applied the technique from Section 3.1.2 to write custom assembly routines to further reduce JEDI's overhead. As a direct result of our techniques from Chapter 3, JEDI's encryption and integrity solutions are capable of running on embedded devices with strict energy and resource constraints, making it suitable for the Internet of Things.

Chapter 8

Related Work

This chapter describes previous and concurrent work related to this dissertation's focus on designing and building systems to help achieve expressive cryptography's full potential. We focus here on related work as it applies to the dissertation as a whole; work related to a particular chapter is discussed at the end of that chapter. Throughout this chapter, our goal is *not* to be exhaustive; we simply aim to provide *examples* of how prior work relates to this dissertation. Specifically, we aim to show (1) how existing work can be seen as applying our techniques from Chapter 3, and (2) how our techniques could potentially be applied to systems in existing work to provide additional benefit. In doing so, our purpose is to demonstrate the utility and relevance of our techniques in Chapter 3 beyond the four systems that we presented in the previous four chapters.

8.1 Other Systems that Exemplify Our System Design Principles

This section describes related work that exemplifies our system design principles set forth in Chapter 3.

8.1.1 Messaging and Storage Systems

A classic technique used in prior systems, particularly in messaging and storage systems, is *hybrid encryption*. The technique in Section 3.2.1 can be seen as a generalization of this idea. With hybrid encryption, instead of encrypting an entire message with public-key cryptography, one can (1) randomly sample a symmetric key, and (2) encrypt the message with the symmetric key, and (3) encrypt the symmetric key with public-key cryptography. This approach is widely deployed, for example, in protocols like TLS and PGP. Over time, this basic approach has been adapted to other settings. For example, Sieve [468] applies it to Attribute-Based Encryption (ABE), using a specialized symmetric-key encryption scheme that enables in situ re-encryption of user data.

The same idea has been applied in the context of cryptographic protocols other than encryption. For example, the authenticated broadcast protocol TESLA [374] applies a similar idea to digital

signatures, enabling one to cheaply extend a digital signature using cryptographic hashes. In the blockchain space, a system that wishes to leverage a blockchain's integrity guarantees may store the *hash* of a document, instead of the document outright, on the blockchain [482]. Another example in the blockchain space is commit chains, which handle transactions off-chain and commit them to the blockchain later, using the blockchain rarely and off of the critical path [271].

μTESLA [375] makes TESLA practical for resource-constrained embedded sensing devices. To do so, one of its techniques is to change the symmetric key *once per epoch* instead of once per message as in TESLA, tying key disclosure to a tunable, time-based epoch rather than to messages sent. This is an example of the technique in Section 3.2.2, and it is similar to using coarser-grained expiry times in JEDI or posting a checkpoint to the blockchain once per epoch in Ghostor. By changing the symmetric key using an epoch of tunable duration, μTESLA trades off the delay after which the receiver can verify a message's authenticity for lower cryptographic costs.

8.1.2 Cryptographic Planners

A number of systems are designed with a component that chooses, for a particular workload, which cryptographic tools to use and how to apply them. We refer to this component as a *cryptographic planner*. Cryptographic planners enable these systems to perform well across a variety of workloads, by dynamically planning how to use cryptography according to the workloads that arrive at runtime.

Cryptographic planners can be seen as a mechanism to apply our techniques for designing systems that use expressive cryptography (Section 3.2). More concretely, the cryptographic planner might, for a given workload and desired security guarantees, determine how to use the chosen cryptographic tools as rarely and cheaply as possible (Section 3.2.1) or determine the cheapest possible cryptographic tools to use in the first place (Section 3.2.3). The "design," however, should be understood as being performed, at least partially, by the *cryptographic planner* at *runtime* when the workload arrives. As a result, cryptographic planners can also be understood as applying the technique from Section 3.1.1—the planners analyze the structure of the computation to decide how and when to use cryptography. This brings the most value when targeting a broad range of workloads, where (1) the "best" cryptographic design may vary considerably from one workload to another, and (2) the workloads are so diverse that it would be too labor-intensive for a cryptography expert to manually determine the best design for each one.

We note that, while MAGE also uses a planner and applies the technique from Section 3.1.1, MAGE's planner is not a cryptographic planner in the sense described here. The key difference is that MAGE's planner decides how to manage memory to support the application, not which cryptographic tools to use or when to use those cryptographic tools.

We now describe systems built with cryptographic planners in greater depth.

8.1.2.1 Planners for Using a Chosen Cryptographic Tool Minimally

SMCQL [38], Conclave [462], and Senate [378] support analytical SQL queries over datasets federated over multiple parties. Each of these systems uses a planner to determine, for a particular

choice of SMPC scheme, how to use it as minimally and cheaply as possible. This can be seen as an application of the technique in Section 3.2.1.

A naïve approach to supporting analytical SQL over a federated dataset would be to use an SMPC protocol to execute the entire SQL query. SMCQL and Conclave observe that it is possible to securely execute an analytical SQL query by applying SMPC to only *part* of the query execution, and use cryptographic planners to minimize the use of SMPC. For example, it may be possible to split query execution into two phases, where the first phase can be run in plaintext and SMPC is only needed in the second phase. SMCQL and Conclave use cryptographic planners to analyze SQL queries and identify which parts of query execution can be run in plaintext, without SMPC.

Senate focuses on federated data analytics problems involving $n > 2$ parties in a malicious threat model. It observes that, rather than executing the entire computation using an n-party SMPC protocol, it can be more efficient to execute parts of the computation with fewer-party SMPC, minimizing the amount of computation involving n-party SMPC. For example, to compute a 4-way join across four parties' datasets, it is more efficient for two pairs of parties to each compute joins over their datasets using 2-party SMPC, followed by a short 4-party SMPC step to combine those results, than to execute the entire join using a monolithic 4-party SMPC step. Senate proposes a planning algorithm to identify such opportunities.

8.1.2.2 Planners for Determining the Cheapest Cryptographic Tools

Systems like Cerebro [514], EzPC [103], and Silph [108] aim to support SMPC workloads that are not necessarily natural to express as SQL queries (e.g., collaborative machine learning). These systems use planners to determine, for a given workload, which cryptographic tools to use so that the workload can execute as efficiently as possible within the required security guarantees. This can be seen as an application of the technique in Section 3.2.3.

The performance of SMPC protocols varies according to not only the expressivity-efficiency trade-off (Section 2.2.2), but also the function f that the parties choose to compute (i.e., the workload). For example, SMPC protocols that model f as an arithmetic circuit over integers in a Galois field (e.g., SPDZ [133]) and SMPC protocols that model f as a boolean circuit (e.g., Yao's Garbled Circuits [496]) may each be preferable for different sets of workloads. Cerebro's compiler plays the role of a cryptographic planner, analyzing the workload and deciding which SMPC protocol to use. This requires the compiler to have an accurate cost model so that it can estimate, for a given workload, which SMPC protocol would be cheapest to use; another related work, CostCO [165], provides a framework for developing such a cost model. Cerebro's compiler also applies techniques similar to SMCQL and Conclave (e.g., identifying parts of the computation that parties can compute locally in plaintext). Additionally, Cerebro focuses on collaborative machine learning applications, allowing an end-to-end approach including support for release policies and auditing for a trained model.

EzPC and Silph take this approach a step further. While Cerebro is limited to choosing a single SMPC protocol for the entire computation, EzPC and Silph can use different SMPC protocols for different parts of a single program. This approach depends on two important factors. First, it depends on the existence of generic, *hybrid* SMPC protocols like ABY [139] that allow one to

securely and efficiently switch between different primitive SMPC protocols. Second, it requires an increased degree of sophistication from the planner, as it must take into account the cost of switching between different SMPC protocols.

8.1.3 Performance-Oriented Systems

As described in Section 2.2, expressive cryptography consumes a wide range of computing resources, including CPU, memory, and network bandwidth. In general, any systems work that improves the efficiency or performance with which an application can make use of these computing resources benefits expressive cryptography. This can be seen as applying the technique in Section 3.2.4.

There is a vast literature on systems that improve efficiency or performance of using computing resources—resource management, after all, is central to the field of computer systems. As a concrete example, we Skyplane [254] is a tool that helps cloud users navigate the trade-off between price and performance for data transfer in the cloud. While Skyplane focuses on the use case of transferring data from the object store (e.g., Amazon S3) in one cloud region to the object store in another cloud region, its core techniques—to use cloud regions as nodes in an overlay network, together with using parallel TCP connections and cloud VMs—applies broadly to data transfer, including to data transfer needed for SMPC. Skyplane's techniques apply naturally to constant-round SMPC protocols like Yao's Garbled Circuits, which involve transferring significant amounts of data between the two parties, when deployed in a cloud setting.

8.2 Applying Our System Design Principles to Other Systems

Our techniques set forth in Chapter 3 are applicable beyond just the four systems presented in this dissertation. This section demonstrates this by explaining how our techniques can be applied to existing systems that use expressive cryptography.

8.2.1 Applicability of Techniques for Supporting Expressive Cryptography

The techniques in Section 3.1, which apply to systems that *support* expressive cryptography, are broadly applicable to cryptography-based applications and systems. The reason is that they apply to a lower layer of the system stack (Figure 3.2) than expressive cryptography and are complementary to the way in which the application makes use of expressive cryptography. For example, as explained in Section 3.3, a platform for collaborative data analytics, such as SMCQL or Conclave, could directly benefit from the techniques from MAGE; while the planners in SMCQL and Conclave minimize the application's use of SMPC, MAGE allows the application's remaining use of SMPC to be as efficient as possible. Similarly, using techniques from Skyplane (Section 8.1.3) to improve network bandwidth performance would benefit Yao's Garbled Circuits in a generic way, regardless of the workload that is run with that SMPC protocol.

In some cases, techniques for better supporting expressive cryptography are complementary and can be used together. For example, DIZK [491] is a system that distributes the generation of a zero-knowledge proof across multiple machines. Its techniques fit in the lower "System" layer in Figure 3.2. Yet DIZK could also benefit from other kinds of system-level techniques for supporting expressive cryptography. For example, memory programming techniques from MAGE could potentially benefit DIZK by allowing the computation on each machine to exceed the available memory, making zero-knowledge proof generation problems of a given size practical with fewer computing resources. That said, applying memory programming to zero-knowledge proof generation may not be straightforward, as zero-knowledge proof generation is not necessarily oblivious—it may require on-the-fly planning with a fast planning algorithm, or potentially manual adaptation to the structure of the particular zero-knowledge proof generation algorithm.

8.2.2 Applicability of Techniques for Using Expressive Cryptography

While JEDI provides end-to-end encryption for publish-subscribe IoT systems using WKD-IBE, alternative designs are possible. For example, PICADOR [75], which we described in Section 7.7, is a publish-subscribe system that uses a different kind of cryptography called Proxy Re-Encryption (PRE) [58, 23]. In PICADOR, publishers encrypt messages according to their own public key. PRE allows the broker, using *re-encryption keys*, to securely re-encrypt those messages to the subscribers' public keys without the broker learning the message. To demonstrate the wide applicability of our techniques in Section 3.2, we demonstrate how we can apply them to a system like PICADOR, just as we did to JEDI and Ghostor.

Given that PRE operations (encryption, decryption, re-encryption) are relatively expensive, our first step is to apply the technique from Section 3.2.1 to use PRE rarely. A natural way to do this is to use hybrid encryption. A publisher encrypts a symmetric key with PRE, sends it to the broker, and then encrypts the actual messages with that symmetric key. The broker stores the encrypted symmetric key. When a subscriber joins the system and needs to access the publisher's messages, the broker re-encrypts the encrypted symmetric key to the subscriber, who can then decrypt the symmetric key and use it decrypt the messages. This technique allows the system to avoid using PRE for most messages.

While this design makes the system more efficient by reducing the frequency with which PRE operations must be invoked, it still requires fresh symmetric keys to be sampled and PRE to be invoked whenever a new user is granted permissions (since they should not be able to see old messages) or a user is revoked access (since they should not be able to see new messages). As a result, PRE-related costs may be high in a system with many users where permissions change frequently. If this is an issue, it can potentially be mitigated by applying the technique from Section 3.2.2. The idea is to rotate keys on a time-based schedule rather than an event-based schedule. For example, a publisher may place an upper bound on the frequency with which it changes symmetric keys (for example, changing symmetric keys no more frequently than once per five minutes). This would add a delay to when changes in permissions take effect, but it could potentially reduce the cost of PRE in environments where permissions are changed frequently. As the delay is tunable, it could potentially be chosen according to resources available to devote

to PRE. For example, in an IoT environment where publishers are constrained in CPU time and energy budget, a longer delay may be appropriate.

The PICADOR authors mention that, as an extension to their system, one could leverage the homomorphic properties of their PRE scheme to compute directly on published data in encrypted form [75]. A downside to applying our techniques to PICADOR as described above is that the resulting design cannot be extended in this way due to its use of symmetric-key cryptography.

Chapter 9

Conclusion

This chapter describes the real-world impact and usage of the systems we built, outlines directions for future work, and summarizes the key points of this dissertation.

9.1 Impact

Some of the work in this dissertation has been adopted by industry or used in deployments to collect and secure real-world data. This section describes the real-world impact that the work in this dissertation has had.

9.1.1 Integration of *TCPlp* into Thread and OpenThread

Both our conclusions in Chapter 5 and our implementation, *TCPlp*, have had impact in the Thread-/OpenThread industrial ecosystem.

Thread [450] is a standard for low-power wireless mesh networks designed for smart home products. It is developed by a consortium of companies in the IoT space, including Google, Apple, Qualcomm, and others [449], and it is used in a variety of products sold by these companies [448]. Originally, Thread specified support for IPv6, UDP, and UDP-based protocols like CoAP, but not TCP. This was consistent with the widely held perception before our work in Chapter 5 that IPv6 has merit in LLNs, but that TCP is not capable of working within the constraints of LLNs and their associated resource-constrained platforms.

The conclusions of Chapter 5, that TCP is capable of running in LLNs (Section 5.9) and brings value to LLN applications, significantly influenced the Thread network standard. The Thread 1.3.0 White Paper [231] notes the benefits of TCP for applications requiring bulk data transfer, as we observed in Section 5.3. Furthermore, as described in the white paper, Thread 1.3.0 requires Thread Certified Components to implement TCP and provide a well-defined API to use it, and provides recommendations for supporting TCP in a Thread network. Some of these recommendations, such as guidance on the MSS mentioned in the white paper, were informed by our study in Chapter 5.

The OpenThread network stack [360] is the leading open-source implementation of the Thread network standard. Originally, OpenThread did not support TCP, but OpenThread adopted *TCPlp* as its TCP stack. Because *TCPlp* is based on the FreeBSD TCP implementation, it benefits from the maturity and full-scale TCP features that *TCPlp* inherits from the FreeBSD implementation (Section 5.5.1). It also benefits from our modifications in Section 5.5 that reduce its memory footprint for low-power embedded devices. As OpenThread is used in smart home products currently on the market [360], *TCPlp*'s adoption in OpenThread paves the way for its deployment in commercial smart home products.

9.1.2 Integration of JEDI into WAVE and WAVEMQ

As described in Section 7.5, JEDI can be applied to bw2. The successor systems to bw2 are WAVE [17], the successor to bw2's authorization layer, and WAVEMQ [13, Chapter 8], the successor to bw2's syndication layer. We integrated JEDI into WAVE and WAVEMQ, adding support for including JEDI keys in WAVE's attestations and securing publish-subscribe communication in WAVEMQ with JEDI.

An example application of WAVE and WAVEMQ is XBOS [170], a microservice-oriented extensible operating system for the built environment. Although early versions of XBOS were based on bw2, XBOS later adopted WAVE for authorization and WAVEMQ as the platform for communication among devices and services [13, Section 7.2]. XBOS was deployed in approximately 20 buildings in California to manage and collect data from smart devices in the built environment. Once we integrated JEDI into WAVEMQ, JEDI was used for secure data collection from a subset of those buildings.

9.2 Future Research Directions

We envision a future in which expressive cryptography can be used pervasively. This would not only bring stronger security to existing applications, but also enable exciting new applications. For example, widespread use of SC would enable organizations (e.g., banks, hospitals) to routinely compute collaboratively on data that they otherwise cannot share due to privacy concerns. We discuss opportunities for additional research to work toward this future, classified according to the two approaches from Chapter 3.

9.2.1 Future Systems that Support Expressive Cryptography

Expressive cryptographic tools increasingly consume *multiple types* of computing resources. For example, SMPC is simultaneously CPU-, memory-, and network-intensive. Efficiently supporting such components (Section 3.1) requires systems thinking to manage and balance the use of *all* of these resources.

9.2.1.1 Memory Programming

As discussed in Section 4.4 and Section 4.6.2.1, MAGE requires SC applications to be rewritten in MAGE's framework. A natural question is: Can we allow cryptographic applications to benefit from memory programming without manually rewriting them? Ideally, memory programming would be as easy to use as `valgrind`. Just as one can check for undefined behavior in a program `a.out` by running `valgrind ./a.out`, one should be able to enable memory programming by running `mage ./a.out`, with only minor code changes to the application to support this.

MAGE's techniques could also be applied to systems not based on expressive cryptography. For example, some plaintext applications, like neural network inference and certain linear algebra workloads (e.g., matrix multiplication), are also oblivious. 3PO [87] explores applying similar ideas to such workloads in the context of memory disaggregation and far memory; a natural follow-up question is how the results may be different for such workloads in the context of paging to an SSD, as in MAGE. Programs designed for hardware enclaves, like Intel SGX, are sometimes written to be oblivious to avoid leaking sensitive information through side channels. Such programs could also be a good fit for applying MAGE's techniques.

9.2.1.2 Networking for SMPC

Mutually distrustful parties using SMPC may be hosted in geographically separate infrastructures (e.g., different cloud providers in different geographic regions). How can we support high-performance network transfer between geographically separate parties? As described in Section 8.1.3, Skyplane [254], which uses cloud-aware network overlays to optimize cloud bulk transfers, is a natural starting point. The research lies in generalizing Skyplane to accelerate *multi-round* SMPC protocols, considering both latency and throughput.

9.2.1.3 Execution Frameworks for zkSNARKs

zkSNARKs have drawn interest due to their applications to decentralized finance and privacy-preserving blockchains. zkSNARK generation is both compute- and memory-intensive, surfacing an opportunity to bring systems expertise to the design of zkSNARK frameworks. The opportunity to leverage GPUs and FPGAs for parts of the computation makes the system design space particularly rich. For example, it may be possible to apply memory programming techniques to optimize data transfer among the memories of CPUs, GPUs, and FPGAs, and to page out to storage as needed.

9.2.2 Future Systems that Use Expressive Cryptography

As applications grow increasingly feature-rich and complex, we must use cryptography expressive enough to capture applications' rich functionality and enable cryptographic security. Achieving good performance, however, requires systems thinking to use expressive cryptography carefully and as efficiently as possible (Section 3.2).

9.2.2.1 Cryptographic Access Control for Data Lakes

To support modern data analytics, many organizations have transitioned away from the classic "data warehouse" model in favor of "data lakes" [20]. Data lakes decouple computing and storage [20], enabling new opportunities for achieving cryptographic security. Just as we did in JEDI, it is natural to consider bringing end-to-end encryption with cryptographically enforced access control to the data lake model, ensuring that data scientists can decrypt only what they are allowed to access. The challenge is to allow access to be specified as *data-dependent views*, as unencrypted systems do. Expressive cryptographic schemes like predicate encryption and functional encryption can capture this model but are heavyweight. A promising approach is to rely on fast block ciphers (e.g., AES) for the bulk of the design, falling back to expressive cryptography only for views that demand particular functionality and security.

9.2.2.2 Applications Using Secure Neural Network Inference

As described in Section 2.3.2, specialized SMPC algorithms have been developed for privacy-preserving neural network inference [346, 260, 397, 291, 345, 392, 306, 356]. A natural question is whether *applications* that need secure neural network inference can be optimized to make use of cryptography in a more efficient way. For example, some applications may use a model obtained by taking an open-source model and fine-tuning it using proprietary data. Depending on how the fine-tuning is done, it may be possible to protect the privacy of such models during inference more efficiently than for models trained from the beginning using proprietary data. Intuitively, this may be the case because the original open-source model is public and only the changes made in fine-tuning based on proprietary data must be hidden.

9.3 Summary

In this dissertation, we described expressive cryptography and explained its potential to enable transformative new applications. For example, Secure Multi-Party Computation (SMPC), a type of expressive cryptography, enables secure collaborative data analytics. This allows multiple hospitals to combine their sensitive patient datasets for research, competing banks to combine their transaction datasets to look for fraud, and regulators to aggregate sensitive data (e.g., data on wages or economic diversity) to benefit society. The technology is receiving some adoption (e.g., by Meta and Google in the advertising space) due to its transformative potential.

Despite its transformative potential, SMPC, and expressive cryptography more broadly, have only been adopted in incipient and isolated use cases. A significant reason for this is that expressive cryptography can be very expensive—it is much slower than regular cryptography and consumes more computing resources (e.g., more CPU time, more RAM, more network bandwidth, etc.). For example, researchers who tried applying SMPC to data analytics tasks have found that it does not scale beyond a few thousand input records because it runs out of memory [462]. In short, the high resource overheads of expressive cryptography are preventing it from reaching its full transformative potential.

This dissertation shows how to use **system design** to allow expressive cryptography to reach its full potential. We take two high-level approaches to achieve this. First, we redesign the underlying systems that expressive cryptography uses. Second, we rethink how and when applications should make use of expressive cryptography. Based on these two high level approaches, we propose, in Chapter 3, six system design techniques for making expressive cryptography efficient. In the next four chapters, we design and implement four different systems using these techniques: MAGE, *TCPlp*, Ghostor, and JEDI. We validate the effectiveness of our six proposed techniques by demonstrating that the resulting systems perform well and can make applications built on expressive cryptography more efficient. In Chapter 8, we present further evidence for the effectiveness of our techniques by pointing out how the designs of existing systems can be understood as applications of our techniques, and demonstrating how other existing systems could potentially benefit from our techniques. Finally, this chapter describes the real-world impact of the systems we built and how our techniques can guide future work on enabling expressive cryptography to reach its full potential.

By making expressive cryptography more efficient, we hope to enable computer users to benefit from the transformative new applications that expressive cryptography enables. A useful analogy, with which we began Chapter 1, is the development and deployment of public-key cryptography in the 1980s and 1990s. Public-key cryptography enabled widespread adoption of applications like e-commerce, telehealth, and end-to-end encrypted messaging. We believe that expressive cryptography can be as transformative as public-key cryptography, enabling widespread adoption of applications that were not previously possible. By helping expressive cryptography reach its full potential, we hope to enable computer users to benefit from the stronger security and more expressive functionality afforded by the resulting computer systems based on expressive cryptography. This, in turn, will help enable cryptographic security for all devices, all applications, and, ultimately, for all people.

Bibliography

[1] Michel Abdalla, Mihir Bellare, Dario Catalano, Eike Kiltz, Tadayoshi Kohno, Tanja Lange, John Malone-Lee, Gregory Neven, Pascal Paillier, and Haixia Shi. "Searchable Encryption Revisited: Consistency Properties, Relation to Anonymous IBE, and Extensions". In: *CRYPTO*. Springer, 2005.

[2] Michel Abdalla, Eike Kiltz, and Gregory Neven. "Generalized Key Delegation for Hierarchical Identity-Based Encryption". In: *ESORICS*. Springer, 2007.

[3] Alexander Afanasyev, Neil Tilley, Peter Reiher, and Leonard Kleinrock. "Host-to-Host Congestion Control for TCP". In: *IEEE Communications Surveys & Tutorials* 12.3 (2010).

[4] Selim G. Akl and Peter D. Taylor. "Cryptographic solution to a problem of access control in a hierarchy". In: *TOCS* 1.3 (1983).

[5] Muhammad Mahbub Alam and Choong Seon Hong. "CRRT: Congestion-Aware and Rate-Controlled Reliable Transport in Wireless Sensor Networks". In: *IEICE Transactions on Communications* 92.1 (2009).

[6] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. "Data Center TCP (DCTCP)". In: *SIGCOMM*. ACM, 2010.

[7] Mark Allman. "TCP Byte Counting Refinements". In: *SIGCOMM-CCR* 29.3 (1999).

[8] Mark Allman. *TCP Congestion Control with Appropriate Byte Counting (ABC)*. RFC 3465. 2003.

[9] Mark Allman, Hari Balakrishnan, and Sally Floyd. *Enhancing TCP's Loss Recovery Using Limited Transmit*. RFC 3042. 2001.

[10] Mark Allman, Dan Glover, and Luis A. Sanchez. *Enhancing TCP Over Satellite Channels using Standard Mechanisms*. RFC 2488. 1999.

[11] Mark Allman and Vern Paxson. "On Estimating End-to-End Network Path Properties". In: *SIGCOMM*. ACM, 1999.

[12] Mark Allman, Vern Paxson, and Ethan Blanton. *TCP Congestion Control*. RFC 5681. 2009.

[13] Michael P Andersen. "Decentralized Authorization with Private Delegation". PhD thesis. University of California, Berkeley, 2019.

[14] Michael P Andersen, Gabe Fierro, and David E. Culler. "System Design for a Synergistic, Low Power Mote/BLE Embedded Platform". In: *IPSN*. IEEE, 2016.

[15] Michael P Andersen, Hyung-Sin Kim, and David E. Culler. "Demo Abstract: Hamilton - A Cost-Effective, Low Power Networked Sensor for Indoor Environment Monitoring". In: *BuildSys*. ACM, 2017.

[16] Michael P Andersen, John Kolb, Kaifei Chen, David E. Culler, and Randy Katz. "Democratizing Authority in the Built Environment". In: *BuildSys*. ACM, 2017.

[17] Michael P Andersen, Sam Kumar, Moustafa AbdelBaky, Gabe Fierro, John Kolb, Hyung-Sin Kim, David E. Culler, and Raluca Ada Popa. "WAVE: A Decentralized Authorization Framework with Transitive Delegation". In: *USENIX Security*. USENIX, 2019.

[18] Edward Arens, Ali Ghahramani, Richard Przybyla, Michael Andersen, Syung Min, Therese Peffer, Paul Raftery, Megan Zhu, Vy Luu, and Hui Zhang. "Measuring 3D indoor air velocity via an inexpensive low-power ultrasonic anemometer". In: *Energy and Buildings* 211 (2020).

[19] Pandarasamy Arjunan, Nipun Batra, Haksoo Choi, Amarjeet Singh, Pushpendra Singh, and Mani B. Srivastava. "SensorAct: A Privacy and Security Aware Federated Middleware for Building Management". In: *BuildSys*. ACM, 2012.

[20] Michael Armbrust, Ali Ghodsi, Reynold Xin, and Matei Zaharia. "Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics". In: *CIDR*. CIDR, 2021.

[21] Mikhail J. Atallah, Marina Blanton, Nelly Fazio, and Keith B. Frikken. "Dynamic and Efficient Key Management for Access Hierarchies". In: *TISSEC* 12.3 (2009).

[22] Mikhail J. Atallah, Marina Blanton, and Keith B. Frikken. "Incorporating temporal capabilities in existing key management schemes". In: *ESORICS*. Springer, 2007.

[23] Giuseppe Ateniese, Kevin Fu, Matthew Green, and Susan Hohenberger. "Improved Proxy Re-Encryption Schemes with Applications to Secure Distributed Storage". In: *NDSS*. Internet Society, 2005.

[24] Nuttapong Attrapadung and Hideki Imai. "Conjunctive Broadcast and Attribute-Based Encryption". In: *ICPBC*. Springer, 2009.

[25] Ahmed Ayadi, Patrick Maillé, and David Ros. "TCP over low-power and lossy networks: tuning the segment size to minimize energy consumption". In: *NTMS*. IEEE, 2011.

[26] Ahmed Ayadi, Patrick Maillé, David Ros, Laurent Toutain, and Tiancong Zheng. "Implementation and Evaluation of a TCP Header Compression for 6LoWPAN". In: *IWCMC*. IEEE, 2011.

[27] Ahmed Ayadi, David Ros, and Laurent Toutain. *TCP header compression for 6LoWPAN*. Tech. rep. draft-aayadi-6lowpan-tcphc-01. Internet Engineering Task Force, 2010. URL: `https://datatracker.ietf.org/doc/draft-aayadi-6lowpan-tcphc/01/`.

[28] Asaph Azaria, Ariel Ekblaw, Thiago Vieira, and Andrew Lippman. "MedRec: Using Blockchain for Medical Data Access and Permission Management". In: *OBD*. IEEE, 2016.

[29] Emmanuel Baccelli, Cenk Gündoğan, Oliver Hahm, Peter Kietzmann, Martine S. Lenders, Hauke Petersen, Kaspar Schleiser, Thomas C. Schmidt, and Matthias Wählisch. "RIOT: an Open Source Operating System for Low-end Embedded Devices in the IoT". In: *IEEE Internet of Things Journal* 5.6 (2018).

[30] Adam Back. *Hashcash - A Denial of Service Counter-Measure*. `http://hashcash.org/hashcash.pdf`. 2002.

[31] Michael Backes, Amir Herzberg, Aniket Kate, and Ivan Pryvalov. "Anonymous RAM". In: *ESORICS*. Springer, 2016.

[32] Randy Baden, Adam Bender, Neil Spring, Bobby Bhattacharjee, and Daniel Starin. "Persona: An Online Social Network with User-Defined Privacy". In: *SIGCOMM*. ACM, 2009.

[33] Hari Balakrishnan. "Challenges to Reliable Data Transport over Heterogeneous Wireless Networks". PhD thesis. University of California, Berkeley, 1998.

[34] Hari Balakrishnan, Venkata N. Padmanabhan, and Randy H. Katz. "The Effects of Asymmetry on TCP Performance". In: *MobiCom*. ACM, 1997.

[35] Hari Balakrishnan, Venkata N. Padmanabhan, Srinivasan Seshan, and Randy H. Katz. "A Comparison of Mechanisms for Improving TCP Performance over Wireless Links". In: *IEEE/ACM Transactions on Networking* 5.6 (1997).

[36] Hari Balakrishnan, Srinivasan Seshan, Elan Amir, and Randy H. Katz. "Improving TCP/IP Performance over Wireless Networks". In: *MobiCom*. ACM, 1995.

[37] Razvan Barbulescu and Sylvain Duquesne. "Updating Key Size Estimations for Pairings". In: *Journal of Cryptology* 32 (2019).

[38] Johes Bater, Gregory Elliott, Craig Eggen, Satyender Goel, Abel Kho, and Jennie Rogers. "SMCQL: Secure Querying for Federated Databases". In: *VLDB* 10.6 (2017).

[39] Andrew Baumann, Marcus Peinado, and Galen Hunt. "Shielding applications from an untrusted cloud with Haven". In: *OSDI*. USENIX, 2014.

[40] Donald Beaver, Silvio Micali, and Phillip Rogaway. "The Round Complexity of Secure Protocols". In: *STOC*. ACM, 1990.

[41] L. A. Belady. "A study of replacement algorithms for virtual storage computers". In: *IBM Syst. J.* 5.2 (1966).

[42] L. A. Belady, R. A. Nelson, and G. S. Shedler. "An Anomaly in Space-Time Characteristics of Certain Programs Running in a Paging Machine". In: *CACM* 12.6 (1969).

[43] Sana Belguith, Shujie Cui, Muhammad Rizwan Asghar, and Giovanni Russello. "Secure Publish and Subscribe Systems with Efficient Revocation". In: *SAC*. ACM, 2018.

[44] Mihir Bellare, Alexandra Boldyreva, Anand Desai, and David Pointcheval. "Key-Privacy in Public-Key Encryption". In: *ASIACRYPT*. Springer, 2001.

[45] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. "Efficient Garbling from a Fixed-Key Blockcipher". In: *S&P*. IEEE, 2013.

[46] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. "Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation". In: *STOC*. ACM, 1988.

[47] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. "Zerocash: Decentralized Anonymous Payments from Bitcoin". In: *S&P*. IEEE, 2014.

[48] Juan Benet. *IPFS - Content Addressed, Versioned, P2P File System*. 2014. arXiv: 1407.3561 [cs.NI].

[49] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. "Lightweight Remote Procedure Call". In: *SOSP*. ACM, 1989.

[50] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. "DepSky: Dependable and Secure Storage in a Cloud-of-Clouds". In: *EuroSys*. ACM, 2011.

[51] John Bethencourt, Amit Sahai, and Brent Waters. "Ciphertext-Policy Attribute-Based Encryption". In: *S&P*. IEEE, 2007.

[52] August Betzler, Carles Gomez, Ilker Demirkol, and Josep Paradells. "CoAP congestion control for the Internet of Things". In: *IEEE Communications Magazine* 54.7 (2016).

[53] Osman Biçer. "Efficiency Optimizations on Yao's Garbled Circuits and Their Practical Applications". Chapters 3 and 4. MA thesis. Istanbul Şehir University, 2017.

[54] Arnar Birgisson, Joe Gibbs Politz, Úlfar Erlingsson, Ankur Taly, Michael Vrable, and Mark Lentczner. "Macaroons: Cookies with Contextual Caveats for Decentralized Authorization in the Cloud". In: *NDSS*. Internet Society, 2014.

[55] *BitInfoCharts*. https://bitinfocharts.com/zcash/.

[56] Erik-Oliver Blass and Florian Kerschbaum. "Private Collaborative Data Cleaning via Non-Equi PSI". In: *S&P*. IEEE, 2023.

[57] Matt Blaze. "A Cryptographic File System for UNIX". In: *CCS*. ACM, 1993.

[58] Matt Blaze, Gerrit Bleumer, and Martin Strauss. "Divertible Protocols and Atomic Proxy Cryptography". In: *EUROCRYPT*. Springer, 1998.

[59] Bluetooth Mesh Working Group. *Mesh Profile 1.0*. https://www.bluetooth.com/specifications/specs/mesh-profile-1-0/. 2017.

[60] Hans Bodlaender, Jens Gustedt, and Jan Arne Telle. "Linear-Time Register Allocation for a Fixed Number of Registers". In: *SODA*. SIAM, 1998.

[61] Alexandra Boldyreva, Vipul Goyal, and Virendra Kumar. "Identity-based Encryption with Efficient Revocation". In: *CCS*. ACM, 2008.

[62] Dan Boneh and Xavier Boyen. “Efficient Selective-ID Secure Identity-Based Encryption Without Random Oracles”. In: *EUROCRYPT*. Springer, 2004.

[63] Dan Boneh, Xavier Boyen, and Eu-Jin Goh. “Hierarchical Identity Based Encryption with Constant Size Ciphertext”. In: *EUROCRYPT*. Springer, 2005.

[64] Dan Boneh, Giovanni Di Crescenzo, Rafail Ostrovsky, and Giuseppe Persiano. “Public Key Encryption with Keyword Search”. In: *EUROCRYPT*. Springer, 2004.

[65] Dan Boneh and Matt Franklin. “Identity-Based Encryption from the Weil Pairing”. In: *CRYPTO*. Springer, 2001.

[66] Dan Boneh, Craig Gentry, and Brent Waters. “Collusion Resistant Broadcast Encryption with Short Ciphertexts and Private Keys”. In: *CRYPTO*. Springer, 2005.

[67] Dan Boneh, Eu-Jin Goh, and Kobbi Nissim. “Evaluating 2-DNF Formulas on Ciphertexts”. In: *TCC*. Springer, 2005.

[68] Dan Boneh, Amit Sahai, and Brent Waters. “Functional Encryption: Definitions and Challenges”. In: *TCC*. Springer, 2011.

[69] Dan Boneh and Brent Waters. “A Fully Collusion Resistant Broadcast, Trace, and Revoke System”. In: *CCS*. ACM, 2006.

[70] Dan Boneh and Brent Waters. “Conjunctive, Subset, and Range Queries on Encrypted Data”. In: *TCC*. Springer, 2007.

[71] Dan Boneh, Brent Waters, and Mark Zhandry. “Low Overhead Broadcast Encryption from Multilinear Maps”. In: *CRYPTO*. Springer, 2014.

[72] Dan Boneh and Mark Zhandry. “Multiparty Key Exchange, Efficient Traitor Tracing, and More from Indistinguishability Obfuscation”. In: *CRYPTO*. Springer, 2014.

[73] Joseph Bonneau. “EthIKS: Using Ethereum to Audit a CONIKS Key Transparency Log”. In: *FC*. Springer, 2016.

[74] Jeff Bonwick. “The Slab Allocator: An Object-Caching Kernel Memory Allocator”. In: *USENIX Summer Technical Conference*. USENIX, 1994.

[75] Cristian Borcea, Arnab “Bobby” Deb Gupta, Yuriy Polyakov, Kurt Rohloff, and Gerard Ryan. “PICADOR: End-to-end encrypted Publish-Subscribe information distribution with proxy re-encryption”. In: *FGCS* 71 (2017).

[76] David Borman, Robert T. Braden, Van Jacobson, and Richard Scheffenegger. *TCP Extensions for High Performance*. RFC 7323. 2014.

[77] Carsten Bormann, Angelo P. Castellani, and Zach Shelby. “CoAP: An Application Protocol for Billions of Tiny Internet Nodes”. In: *IEEE Internet Computing* 16.2 (2012).

[78] Gaetano Borriello and Roy Want. “Embedded Computation Meets The World Wide Web”. In: *CACM* 43.5 (2000).

[79] Raphael Bost, Raluca Ada Popa, Stephen Tu, and Shafi Goldwasser. “Machine Learning Classification over Encrypted Data”. In: *NDSS*. Internet Society, 2015.

[80] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly Media, 2006. Chap. 17, p. 679.

[81] Sean Bowe. *BLS12-381: New zk-SNARK Elliptic Curve Construction*. `https://z.cash/blog/new-snark-curve/`. 2017.

[82] Xavier Boyen. *Expressive Cryptography*. `https://crypto.stanford.edu/~xb/cacr12/index.html`. Dec. 2012.

[83] Xavier Boyen. "Expressive Cryptography: Lattice Perspectives". In: *Australasian Conference on Information Security and Privacy*. Springer, 2013.

[84] Xavier Boyen and Brent Waters. "Anonymous Hierarchical Identity-Based Encryption (Without Random Oracles)". In: *CRYPTO*. Springer, 2006.

[85] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. "Efficient Pseudorandom Correlation Generators: Silent OT Extension and More". In: *CRYPTO*. Springer, 2019.

[86] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. "(Leveled) Fully Homomorphic Encryption without Bootstrapping". In: *ITCS*. ACM, 2012.

[87] Christopher Branner-Augmon, Narek Galstyan, Sam Kumar, Emmanuel Amaro, Amy Ousterhout, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. *3PO: Programmed Far-Memory Prefetching for Oblivious Applications*. 2022. arXiv: `2207.07688 [cs.OS]`.

[88] Davide Brunelli, Ivan Minakov, Roberto Passerone, and Maurizio Rossi. "POVOMON: an Ad-hoc Wireless Sensor Network for Indoor Environmental Monitoring". In: *EESMS*. IEEE, 2014.

[89] F. Buccafurri, G. Lax, S. Nicolazzo, and A. Nocera. "Accountability-Preserving Anonymous Delivery of Cloud services". In: *TrustBus*. Springer, 2015.

[90] Niklas Buescher and Stefan Katzenbeisser. "Faster Secure Computation through Automatic Parallelization". In: *USENIX Security*. USENIX, 2015.

[91] Michael Buettner, Gary V. Yee, Eric Anderson, and Richard Han. "X-MAC: A Short Preamble MAC Protocol for Duty-Cycled Wireless Sensor Networks". In: *SenSys*. ACM, 2006.

[92] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution". In: *USENIX Security*. USENIX, 2018.

[93] Vitalik Buterin. *Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform*. `https://ethereum.org/669c9e2e2027310b6b3cdce6e1c52962/Ethereum_Whitepaper_-_Buterin_2014.pdf`. 2014.

[94] *bw2*. `https://github.com/immesys/bw2`.

[95] Jan L. Camenisch, Jean-Marc Piveteau, and Markus A. Stadler. "Blind Signatures Based on the Discrete Logarithm Problem". In: *EUROCRYPT*. Springer, 1994.

[96] Brad Campbell. *Introducing Hail*. `https://www.tockos.org/blog/2017/introducing-hail/`. 2017.

[97] Ran Canetti, Shai Halevi, and Jonathan Katz. "A Forward-Secure Public-Key Encryption Scheme". In: *EUROCRYPT*. Springer, 2003.

[98] Cape Privacy. *Cape Privacy (Formerly Dropout Labs)—Medium*. `https://medium.com/dropoutlabs`. Accessed: June 21, 2023.

[99] Sergiu Carpov, Paul Dubrulle, and Renaud Sirdey. "Armadillo: A Compilation Chain for Privacy Preserving Applications". In: *SCC*. ACM, 2015.

[100] Angelo P. Castellani, Mattia Gheda, Nicola Bui, Michele Rossi, and Michele Zorzi. "Web Services for the Internet of Things through CoAP and EXI". In: *ICC*. IEEE, 2011.

[101] Miguel Castro and Barbara Liskov. "Practical Byzantine Fault Tolerance". In: *OSDI*. USENIX, 1999.

[102] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. "Register Allocation via Coloring". In: *Computer Languages* 6.1 (1981).

[103] Nishanth Chandran, Divya Gupta, Aseem Rastogi, Rahul Sharma, and Shardul Tripathi. "EzPC: Programmable and Efficient Secure Two-Party Computation for Machine Learning". In: *EuroS&P*. IEEE, 2019.

[104] David Chaum. "Blind Signature System". In: *CRYPTO*. Plenum, 1983.

[105] David Chaum. "Blind Signatures for Untraceable Payments". In: *CRYPTO*. Plenum, 1982.

[106] David Chaum and Eugène van Heyst. "Group Signatures". In: *EUROCRYPT*. Springer, 1991.

[107] David L. Chaum. "Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms". In: *CACM* 24.2 (1981).

[108] Edward Chen, Jinhao Zhu, Alex Ozdemir, Riad Wahby, Fraser Brown, and Wenting Zheng. "Silph: A Framework for Scalable and Accurate Generation of Hybrid MPC Protocols". In: *S&P*. IEEE, 2023.

[109] Hao Chen, Miran Kim, Ilya Razenshteyn, Dragos Rotaru, Yongsoo Song, and Sameer Wagh. "Maliciously Secure Matrix Multiplication with Applications to Private Deep Learning". In: *ASIACRYPT*. Springer, 2020.

[110] Shuo Chen, Rui Wang, XiaoFeng Wang, and Kehuan Zhang. "Side-Channel Leaks in Web Applications: A Reality Today, a Challenge Tomorrow". In: *S&P*. IEEE, 2010.

[111] Weikeng Chen. "Building Cryptographic Systems from Distributed Trust". PhD thesis. University of California, Berkeley, 2022.

[112] Weikeng Chen and Raluca Ada Popa. “Metal: A Metadata-Hiding File-Sharing System”. In: *NDSS*. Internet Society, 2020.

[113] Raymond Cheng, William Scott, Elisaweta Masserova, Irene Zhang, Vipul Goyal, Thomas Anderson, Arvind Krishnamurthy, and Bryan Parno. “Talek: Private Group Messaging with Hidden Access Patterns”. In: *ACSAC*. ACM, 2020.

[114] Jung Hee Cheon. “Security Analysis of the Strong Diffie-Hellman Problem”. In: *EUROCRYPT*. Springer, 2006.

[115] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. “Homomorphic Encryption for Arithmetic of Approximate Numbers”. In: *ASIACRYPT*. Springer, 2017.

[116] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. “Faster Fully Homomorphic Encryption: Bootstrapping in Less Than 0.1 Seconds”. In: *ASIACRYPT*. Springer, 2016.

[117] Mi-Joung Choi, Hong-Taek Ju, Hyun-Jun Cha, Sook-Hyang Kim, and James Won-Ki Hong. “An Efficient Embedded Web Server for Web-based Network Element Management”. In: *NOMS*. IEEE, 2000.

[118] Cisco. *The Internet of Things Reference Model*. 2014.

[119] David D. Clark. “The Structuring of Systems Using Upcalls”. In: *SOSP*. ACM, 1985.

[120] David D. Clark, Van Jacobson, John Romkey, and Howard Salwen. “An Analysis of TCP Processing Overhead”. In: *IEEE Communications Magazine* 27.6 (1989).

[121] Dwaine Clarke, Jean-Emile Elien, Carl Ellison, Matt Fredette, Alexander Morcos, and Ronald L. Rivest. “Certificate chain discovery in SPKI/SDSI”. In: *Journal of Computer Security* 9.4 (2001).

[122] Coinbase. *Coinbase to acquire leading cryptographic security company, Unbound Security*. `https://www.coinbase.com/blog/coinbase-to-acquire-leading-cryptographic-security-company-unbound-security`. Accessed: June 21, 2023.

[123] Walter Colitti, Kris Steenhaut, Niccolò De Caro, Bogdan Buta, and Virgil Dobrota. “Evaluation of Constrained Application Protocol for Wireless Sensor Networks”. In: *LANMAN*. IEEE, 2011.

[124] Stefan Contiu, Sébastien Vaucher, Rafael Pires, Marcelo Pasin, Pascal Felber, and Laurent Réveillère. “Anonymous and Confidential File Sharing over Untrusted Clouds”. In: *SRDS*. IEEE, 2019.

[125] Keith D. Cooper and L. Taylor Simpson. “Live Range Splitting in a Graph Coloring Register Allocator”. In: *CC*. Springer, 1998.

[126] Henry Corrigan-Gibbs. “Protecting Privacy by Splitting Trust”. PhD thesis. Stanford University, 2019.

[127] Henry Corrigan-Gibbs and Dan Boneh. “Prio: Private, Robust and Scalable Computation of Aggregate Statistics”. In: *NSDI*. USENIX, 2017.

[128] Henry Corrigan-Gibbs, Dan Boneh, and David Mazières. "Riposte: An Anonymous Messaging System Handling Millions of Users". In: *S&P*. IEEE, 2015.

[129] Jason Crampton, Naomi Farley, Gregory Gutin, Mark Jones, and Bertram Poettering. "Cryptographic Enforcement of Information Flow Policies Without Public Information". In: *ACNS*. Springer, 2015.

[130] Jason Crampton, Keith Martin, and Peter Wild. "On Key Assignment for Hierarchical Access Control". In: *CSFW*. IEEE, 2006.

[131] Crypho. *Enterprise communications with end-to-end encryption*. `https://www.crypho.com/`.

[132] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. "Practical Covertly Secure MPC for Dishonest Majority – Or: Breaking the SPDZ Limits". In: *ESORICS*. Springer, 2013.

[133] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. "Multiparty Computation from Somewhat Homomorphic Encryption". In: *CRYPTO*. Springer, 2012.

[134] Roshan Dathathri, Blagovesta Kostova, Olli Saarikivi, Wei Dai, Kim Laine, and Madan Musuvathi. "EVA: An Encrypted Vector Arithmetic Language and Compiler for Efficient Homomorphic Computation". In: *PLDI*. ACM, 2020.

[135] Jessica Davis. *The 10 Biggest Healthcare Data Breaches of 2019, So Far*. `https://healthitsecurity.com/news/the-10-biggest-healthcare-data-breaches-of-2019-so-far`. Accessed: September 12, 2019.

[136] Stephen Dawson-Haggerty, Xiaofan Jiang, Gilman Tolle, Jorge Ortiz, and David Culler. "sMAP – a Simple Measurement and Actuation Profile for Physical Information". In: *SenSys*. ACM, 2010.

[137] Stephen Dawson-Haggerty, Andrew Krioukov, Jay Taneja, Sagar Karandikar, Gabe Fierro, Nikita Kitaev, and David Culler. "BOSS: Building Operating System Services". In: *NSDI*. USENIX, 2013.

[138] Deloitte. *Using blockchain to drive supply chain innovation*. `https://www2.deloitte.com/content/dam/Deloitte/us/Documents/process-and-operations/us-blockchain-to-drive-supply-chain-innovation.pdf`. 2017.

[139] Daniel Demmler, Thomas Schneider, and Michael Zohner. "ABY – A Framework for Efficient Mixed-Protocol Secure Two-Party Computation". In: *NDSS*. Internet Society, 2015.

[140] Peter J. Denning. "Thrashing: Its causes and prevention". In: *AFIPS*. ACM, 1968.

[141] Peter J. Denning. "Virtual Memory". In: *CSUR* 2.3 (1970).

[142] Peter J. Denning. "Working Sets Past and Present". In: *IEEE Trans. Softw. Eng.* SE-6.1 (1980).

[143] DigiKey Electronics. *ATSAMR21E18A-MU Microchip Technology*. Accessed: February 8, 2019.

[144] Roger Dingledine, Nick Mathewson, and Paul Syverson. “Tor: The Second-Generation Onion Router”. In: *USENIX Security*. USENIX, 2004.

[145] Yevgeniy Dodis and Nelly Fazio. “Public Key Broadcast Encryption for Stateless Receivers”. In: *DRM*. Springer, 2002.

[146] Peter Druschel and Larry L. Peterson. “Fbufs: A High-Bandwidth Cross-Domain Transfer Facility”. In: *SOSP*. ACM, 1993.

[147] Duality. *Duality Technologies - Secure Data Collaboration Products*. `https://dualitytech.com/`. Accessed: June 21, 2023.

[148] Léo Ducas and Daniele Micciancio. “FHEW: Bootstrapping Homomorphic Encryption in Less Than a Second”. In: *EUROCRYPT*. Springer, 2015.

[149] Paul Duffy. *Beyond MQTT: A Cisco View on IoT Protocols*. `https://blogs.cisco.com/digital/beyond-mqtt-a-cisco-view-on-iot-protocols`. Accessed: September 9, 2018. 2013.

[150] Adam Dunkels. “Full TCP/IP for 8-Bit Architectures”. In: *MobiSys*. USENIX, 2003.

[151] Adam Dunkels, Juan Alonso, and Thiemo Voigt. *Making TCP/IP Viable for Wireless Sensor Networks*. Tech. rep. SICS-T-2003/23-SE. Swedish Institute for Computer Science, 2003. URL: `http://www.diva-portal.org/smash/record.jsf?pid=diva2:1041597`.

[152] Adam Dunkels, Juan Alonso, Thiemo Voigt, Hartmut Ritter, and Jochen Schiller. “Connecting Wireless Sensornets with TCP/IP Networks”. In: *WWIC*. Springer, 2004.

[153] Adam Dunkels, Björn Grönvall, and Thiemo Voigt. “Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors”. In: *LCN*. IEEE, 2004.

[154] Simon Duquennoy, Beshr Al Nahas, Olaf Landsiedel, and Thomas Watteyne. “Orchestra: Robust Mesh Networks Through Autonomously Scheduled TSCH”. In: *SenSys*. ACM, 2015.

[155] Simon Duquennoy, Fredrik Österlind, and Adam Dunkels. “Lossy Links, Low Power, High Throughput”. In: *SenSys*. ACM, 2011.

[156] Mathilde Durvy, Julien Abeillé, Patrick Wetterwald, Colin O’Flynn, Blake Leverett, Eric Gnoske, Michael Vidales, Geoff Mulligan, Nicolas Tsiftes, Niclas Finne, and Adam Dunkels. “Poster Abstract: Making Sensor Networks IPv6 Ready”. In: *SenSys*. ACM, 2008.

[157] Prabal Dutta, David Culler, and Scott Shenker. “Procrastination Might Lead to a Longer and More Useful Life”. In: *HotNets*. ACM, 2007.

[158] Prabal Dutta, Stephen Dawson-Haggerty, Yin Chen, Chieh-Jan Mike Liang, and Andreas Terzis. “Design and Evaluation of a Versatile and Efficient Receiver-Initiated Link Layer for Low-power Wireless”. In: *SenSys*. ACM, 2010.

[159] Michael Egorov, MacLane Wilkison, and David Nuñez. *NuCypher KMS: Decentralized key management system*. 2017. arXiv: `1707.06140 [cs.CR]`.

[160] Adam Eijdenberg, Ben Laurie, and Al Cutter. *Verifiable Data Structures*. https://github.com/google/trillian/blob/master/docs/papers/VerifiableDataStructures.pdf.

[161] Erez Eizenman. *Scotiabank's chief risk officer on the state of anti-money laundering*. https://www.mckinsey.com/capabilities/risk-and-resilience/our-insights/scotiabanks-chief-risk-officer-on-the-state-of-anti-money-laundering. Accessed: June 30, 2023. 2019.

[162] Taher ElGamal. "A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms". In: *CRYPTO*. Springer, 1984.

[163] Deborah Estrin, Ramesh Govindan, John Heidemann, and Satish Kumar. "Next Century Challenges: Scalable Coordination in Sensor Networks". In: *MobiCom*. ACM, 1999.

[164] Kevin Fall and Sally Floyd. "Simulation-based Comparisons of Tahoe, Reno and SACK TCP". In: *SIGCOMM-CCR* 26.3 (1996).

[165] Vivian Fang, Lloyd Brown, William Lin, Wenting Zheng, Aurojit Panda, and Raluca Ada Popa. "CostCo: An automatic cost modeling framework for secure multi-party computation". In: *EuroS&P*. IEEE, 2022.

[166] Martin Farach and Vincenzo Liberatore. "On Local Register Allocation". In: *SODA*. SIAM, 1998.

[167] Ariel J. Feldman, William P. Zeller, Michael J. Freedman, and Edward W. Felten. "SPORC: Group Collaboration using Untrusted Cloud Resources". In: *OSDI*. USENIX, 2010.

[168] Axel Feldmann, Nikola Samardzic, Aleksandar Krastev, Srini Devadas, Ron Dreslinski, Christopher Peikert, and Daniel Sanchez. "F1: A Fast and Programmable Accelerator for Fully Homomorphic Encryption". In: *MICRO*. ACM, 2021.

[169] Mark Christopher Feldmeier. "Personalized Building Comfort Control". PhD thesis. Massachusetts Institute of Technology, 2009.

[170] Gabriel Fierro and David E. Culler. *XBOS: An Extensible Building Operating System*. Tech. rep. UCB/EECS-2015-197. EECS Department, University of California, Berkeley, 2015. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-197.html.

[171] *Filecoin*. https://filecoin.io. Accessed: January 19, 2018.

[172] Fireblocks. *7 Reasons Why MPC Is the Next Generation of Private Key Security*. https://www.fireblocks.com/blog/7-reasons-why-mpc-is-the-next-generation-of-private-key-security/. 2019.

[173] Fireblocks. *MPC Wallet As A Service Technology - Fireblocks*. Accessed: May 29, 2023.

[174] Sally Floyd. *HighSpeed TCP for Large Congestion Windows*. RFC 3649. 2003.

[175] Sally Floyd. "TCP and Explicit Congestion Notification". In: *SIGCOMM-CCR* 24.5 (1994).

[176] Sally Floyd and Van Jacobson. “Random Early Detection Gateways for Congestion Avoidance”. In: *IEEE/ACM Transactions on Networking* 1.4 (1993).

[177] Sadjad Fouladi, John Emmons, Emre Orbay, Catherine Wu, Riad S. Wahby, and Keith Winstein. “Salsify: Low-Latency Network Video Through Tighter Integration Between a Video Codec and a Transport Protocol”. In: *NSDI*. USENIX, 2018.

[178] Michael J. Freedman, Kobbi Nissim, and Benny Pinkas. “Efficient Private Matching and Set Intersection”. In: *EUROCRYPT*. Springer, 2004.

[179] Tilman Frosch, Christian Mainka, Christoph Bader, Florian Bergsma, Jörg Schwenk, and Thorsten Holz. “How Secure is TextSecure?” In: *EuroS&P*. IEEE, 2016.

[180] Jonathan Fürst, Kaifei Chen, Mohammed Aljarrah, and Philippe Bonnet. “Leveraging Physical Locality to Integrate Smart Appliances in Non-Residential Buildings with Ultrasound and Bluetooth Low Energy”. In: *IoTDI*. IEEE, 2016.

[181] William C. Garrison III, Adam Shull, Steven Myers, and Adam J. Lee. “On the Practicality of Cryptographically Enforcing Dynamic Access Control Policies in the Cloud”. In: *S&P*. IEEE, 2016.

[182] Craig Gentry. “Fully Homomorphic Encryption Using Ideal Lattices”. In: *STOC*. ACM, 2009.

[183] Craig Gentry and Shai Halevi. “Hierarchical Identity Based Encryption with Polynomially Many Levels”. In: *TCC*. Springer, 2009.

[184] Craig Gentry, Shai Halevi, and Nigel P. Smart. “Fully Homomorphic Encryption with Polylog Overhead”. In: *EUROCRYPT*. Springer, 2012.

[185] Craig Gentry, Amit Sahai, and Brent Waters. “Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based”. In: *CRYPTO*. Springer, 2013.

[186] Craig Gentry and Alice Silverberg. “Hierarchical ID-Based Cryptography”. In: *ASIACRYPT*. Springer, 2002.

[187] Mario Gerla, Ken Tang, and Rajive Bagrodia. “TCP Performance in Wireless Multi-hop Networks”. In: *WMCSA*. IEEE, 1999.

[188] Seth Gilbert and Nancy Lynch. “Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services”. In: *SIGACT News* 33.2 (2002).

[189] Omprakash Gnawali, Rodrigo Fonseca, Kyle Jamieson, David Moss, and Philip Levis. “Collection Tree Protocol”. In: *SenSys*. ACM, 2009.

[190] Eu-Jin Goh, Hovav Shacham, Nagendra Modadugu, and Dan Boneh. “SiRiUS: Securing Remote Untrusted Storage”. In: *NDSS*. Internet Society, 2003.

[191] Oded Goldreich. *Foundations of Cryptography*. Vol. 1. Cambridge University Press, 2007.

[192] Oded Goldreich, Silvio Micali, and Avi Wigderson. “How to Play Any Mental Game, or A Completeness Theorem for Protocols with Honest Majority”. In: *STOC*. ACM, 1987.

[193] Oded Goldreich and Rafail Ostrovsky. “Software Protection and Simulation on Oblivious RAMs”. In: *J. ACM* 43.3 (1996).

[194] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. “The Knowledge Complexity of Interactive Proof-Systems”. In: *STOC*. ACM, 1985.

[195] Carles Gomez, Andrés Arcia-Moret, and Jon Crowcroft. “TCP in the Internet of Things: From Ostracism to Prominence”. In: *IEEE Internet Computing* 22.1 (2018).

[196] Kazuhiro Gomi. *Multi-Party Computation: Private Inputs, Public Outputs*. `https://www.forbes.com/sites/forbestechcouncil/2021/10/26/multi-party-computation-private-inputs-public-outputs/?sh=7e43d5e51bb0`. 2021.

[197] Kazuhiro Gomi. *Rethinking Encryption To Enhance Security And Utility*. `https://www.forbes.com/sites/forbestechcouncil/2020/06/12/rethinking-encryption-to-enhance-security-and-utility`. Accessed: October 17, 2021. 2020.

[198] Fernando Gont and Andrew Yourtchenko. *On the Implementation of the TCP Urgent Mechanism*. RFC 6093. 2011.

[199] Google Cloud. *Machine families resource and comparison guide — Compute Engine Documentation — Google Cloud*. `https://cloud.google.com/compute/docs/machine-types`.

[200] Sergey Gorbunov, Vinod Vaikuntanathan, and Hoeteck Wee. “Attribute-Based Encryption for Circuits”. In: *STOC*. ACM, 2013.

[201] Sergey Gorbunov, Vinod Vaikuntanathan, and Hoeteck Wee. “Functional Encryption with Bounded Collusions via Multi-party Computation”. In: *CRYPTO*. Springer, 2012.

[202] William Gordon, Aneesh Chopra, and Adam Landman. *Patient-Led Data Sharing—A New Paradigm for Electronic Health Data*. `https://catalyst.nejm.org/patient-led-health-data-paradigm/`. Accessed: September 12, 2019.

[203] Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. “Attribute-Based Encryption for Fine-Grained Access Control of Encrypted Data”. In: *CCS*. ACM, 2006.

[204] Luigi A. Grieco and Saverio Mascolo. “Performance Evaluation and Comparison of Westwood+, New Reno, and Vegas TCP Congestion Control”. In: *SIGCOMM-CCR* 34.2 (2004).

[205] Paul Grubbs, Richard McPherson, Muhammad Naveed, Thomas Ristenpart, and Vitaly Shmatikov. “Breaking Web Applications Built On Top of Encrypted Data”. In: *CCS*. ACM, 2016.

[206] Chun Guo, Jonathan Katz, Xiao Wang, Chenkai Weng, and Yu Yu. “Better Concrete Security for Half-Gates Garbling (in the Multi-instance Setting)”. In: *CRYPTO*. Springer, 2020.

[207] Sangtae Ha, Injong Rhee, and Lisong Xu. “CUBIC: A New TCP-Friendly High-Speed TCP Variant”. In: *ACM SIGOPS Operating Systems Review* 42.5 (2008).

[208] Stuart Haber and W. Scott Stornetta. “How to Time-Stamp a Digital Document”. In: *CRYPTO*. Springer, 1990.

[209] Shai Halevi. “Advanced Cryptography: Promise and Challenges”. In: *CCS*. ACM, 2018.

[210] *Hamilton IoT*. `https://hamiltoniot.com/`. Accessed: November 30, 2018.

[211] Ariel Hamlin, Rafail Ostrovsky, Mor Weiss, and Daniel Wichs. “Private Anonymous Data Access”. In: *EUROCRYPT*. Springer, 2019.

[212] Kieran Harty and David R. Cheriton. “Application-Controlled Physical Memory using External Page-Cache Management”. In: *ASPLOS*. ACM, 1992.

[213] Milad Hashemi, Kevin Swersky, Jamie A. Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, and Parthasarathy Ranganathan. “Learning Memory Access Patterns”. In: *ICML*. ML Research Press, 2018.

[214] Marcella Hastings, Brett Hemenway, Daniel Noble, and Steve Zdancewic. “SoK: General Purpose Compilers for Secure Multi-Party Computation”. In: *S&P*. IEEE, 2019.

[215] Jun He, John Bent, Aaron Torres, Gary Grider, Garth Gibson, Carlos Maltzahn, and Xian-He Sun. “I/O Acceleration with Pattern Detection”. In: *HPDC*. ACM, 2013.

[216] Warren He, Devdatta Akhawe, Sumeet Jain, Elaine Shi, and Dawn Song. “ShadowCrypt: Encrypted Web Applications for Everyone”. In: *CCS*. ACM, 2014.

[217] Tom Henderson, Sally Floyd, Andrei Gurtov, and Yoshifumi Nishida. *The NewReno Modification to TCP’s Fast Recovery Algorithm*. RFC 6582. 2012.

[218] Maurice P. Herlihy and Jeannette M. Wing. “Linearizability: A Correctness Condition for Concurrent Objects”. In: *TOPLAS* 12.3 (1990).

[219] Kasun Hewage, Simon Duquennoy, Venkatraman Iyer, and Thiemo Voigt. “Enabling TCP in Mobile Cyber-physical Systems”. In: *MASS*. IEEE, 2015.

[220] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. “System Architecture Directions for Networked Sensors”. In: *ASPLOS*. ACM, 2000.

[221] Andreas Holzer, Martin Franz, Stefan Katzenbeisser, and Helmut Veith. “Secure Two-Party Computations in ANSI C”. In: *CCS*. ACM, 2012.

[222] Jelle van den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. “Vuvuzela: Scalable Private Messaging Resistant to Traffic Analysis”. In: *SOSP*. ACM, 2015.

[223] David Hoppe. *Blockchain Use Cases: Electronic Health Records*. `https://gammalaw.com/blockchain_use_cases_electronic_health_records/`. Accessed: September 12, 2019.

[224] Jeremy Horwitz and Ben Lynn. “Toward Hierarchical Identity-Based Encryption”. In: *EUROCRYPT*. Springer, 2002.

[225] Yih-Chun Hu, Markus Jakobsson, and Adrian Perrig. “Efficient Constructions for One-Way Hash Chains”. In: *ACNS*. Springer, 2005.

[226] Yuncong Hu, Sam Kumar, and Raluca Ada Popa. “Ghostor: Toward a Secure Data-Sharing System from Decentralized Trust”. In: *NSDI*. USENIX, 2020.

[227] Yuncong Hu, Sam Kumar, and Raluca Ada Popa. *Ghostor: Toward a Secure Data-Sharing System from Decentralized Trust*. Cryptology ePrint Archive, Paper 2020/648. 2020. URL: https://eprint.iacr.org/2020/648.

[228] Hui-Feng Huang and Chin-Chen Chang. "A new cryptographic key assignment scheme with time-constraint access control in a hierarchy". In: *Computer Standards & Interfaces* 26.3 (2004).

[229] Yan Huang, David Evans, and Jonathan Katz. "Private Set Intersection: Are Garbled Circuits Better than Custom Protocols?" In: *NDSS*. Internet Society, 2012.

[230] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. "Faster Secure Two-Party Computation Using Garbled Circuits". In: *USENIX Security*. USENIX, 2011.

[231] Jonathan Hui, Sujata Neidig, and Alan Collins. *Thread 1.3.0 Features White Paper*. https://www.threadgroup.org/Portals/0/documents/support/Thread1.3.0WhitePaper_07192022_3990_1.pdf. 2022.

[232] Jonathan W. Hui. Personal Communication. Sept. 21, 2016.

[233] Jonathan W. Hui and David E. Culler. "IP is Dead, Long Live IP for Wireless Sensor Networks". In: *SenSys*. ACM, 2008.

[234] Bret Hull, Kyle Jamieson, and Hari Balakrishnan. "Mitigating Congestion in Wireless Sensor Networks". In: *SenSys*. ACM, 2004.

[235] Ryan Hurst and Gary Belvin. *Security Through Transparency*. https://security.googleblog.com/2017/01/security-through-transparency.html.

[236] Jakob Hviid and Mikkel Baun Kjærgaard. "Activity-Tracking Service for Building Operating Systems". In: *PerFoT*. IEEE, 2018.

[237] Identity Theft Resource Center. *At Mid-Year, U.S. Data Breaches Increase at Record Pace*. https://www.idtheftcenter.org/post/at-mid-year-u-s-data-breaches-increase-at-record-pace/. 2018.

[238] Heesu Im. "TCP Performance Enhancement in Wireless Networks". PhD thesis. Seoul National University, 2015.

[239] *imix: Low-Power IoT Research Platform*. https://github.com/helena-project/imix. 2017.

[240] Hiroshi Inamura, Gabriel Montenegro, Reiner Ludwig, Andrei Gurtov, and Farid Khafizov. *TCP over Second (2.5G) and Third (3G) Generation Wireless Networks*. RFC 3481. 2003.

[241] Inpher. *Privacy Preserving Machine Learning and Analytics Pioneer — Inpher*. https://inpher.io/. Accessed: June 21, 2023.

[242] Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin. "Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks". In: *MobiCom*. ACM, 2000.

[243] Mihaela Ion, Ben Kreuter, Ahmet Erhan Nergiz, Sarvar Patel, Shobhit Saxena, Karn Seth, Mariana Raykova, David Shanahan, and Moti Yung. "On Deploying Secure Computing: Private Intersection-Sum-with-Cardinality Protocols". In: *EuroS&P*. IEEE, 2020.

[244] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. "Extending Oblivious Transfers Efficiently". In: *CRYPTO*. Springer, 2003.

[245] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. "Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation". In: *NDSS*. Internet Society, 2012.

[246] Davide Italiano and Alexander Motin. "Calloutng: a new infrastructure for timer facilities in the FreeBSD kernel". In: *AsiaBSDCon*. AsiaBSDCon, 2013.

[247] Yogesh G. Iyer, Shashidhar Gandham, and S. Venkatesan. "STCP: A Generic Transport Layer Protocol for Wireless Sensor Networks". In: *ICCCN*. IEEE, 2005.

[248] Van Jacobson. *Compressing TCP/IP Headers for Low-Speed Serial Links*. RFC 1144. 1990.

[249] Van Jacobson. "Congestion Avoidance and Control". In: *SIGCOMM*. ACM, 1988.

[250] Toby Jaffey. *MQTT and CoAP, IoT Protocols*. `https://www.eclipse.org/community/eclipse_newsletter/2014/february/article2.php`. Accessed: September 9, 2018. 2014.

[251] Geetha Jagannathan and Rebecca N. Wright. "Privacy-Preserving Distributed k-Means Clustering over Arbitrarily Partitioned Data". In: *KDD*. ACM, 2005.

[252] Akanksha Jain and Calvin Lin. "Rethinking Belady's Algorithm to Accommodate Prefetching". In: *ISCA*. IEEE, 2018.

[253] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Kurt Keutzer, Ion Stoica, and Joey Gonzalez. "Checkmate: Breaking the Memory Wall with Optimal Tensor Rematerialization". In: *MLSys*. `https://mlsys.org`, 2020.

[254] Paras Jain, Sam Kumar, Sarah Wooders, Shishir G. Patil, Joseph E. Gonzalez, and Ion Stoica. "Skyplane: Optimizing Transfer Cost and Throughput Using Cloud-Aware Overlays". In: *NSDI*. USENIX, 2023.

[255] Somesh Jha, Louis Kruger, and Vitaly Shmatikov. "Towards Practical Privacy for Genomic Computation". In: *S&P*. IEEE, 2008.

[256] Xiaoqian Jiang, Miran Kim, Kristin Lauter, and Yongsoo Song. "Secure Outsourced Matrix Computation and Application to Neural Networks". In: *CCS*. ACM, 2018.

[257] Cheng Jin, David X. Wei, and Steven H. Low. "FAST TCP: Motivation, Architecture, Algorithms, Performance". In: *INFOCOM*. IEEE, 2004.

[258] Sally Johnson. *Constrained Application Protocol: CoAP is IoT's 'modern' protocol*. `https://internetofthingsagenda.techtarget.com/feature/Constrained-Application-Protocol-CoAP-is-IoTs-modern-protocol`. Accessed: September 9, 2018. 2016.

[259] Deokwoo Jung, Zhenjie Zhang, and Marianne Winslett. "Vibration Analysis for IoT Enabled Predictive Maintenance". In: *ICDE*. IEEE, 2017.

[260] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. "GAZELLE: A Low Latency Framework for Secure Neural Network Inference". In: *USENIX Security*. USENIX, 2018.

[261] Mahesh Kallahalla, Erik Riedel, Ram Swaminathan, Qian Wang, and Kevin Fu. "Plutus: Scalable secure file sharing on untrusted storage". In: *FAST*. USENIX, 2003.

[262] Nikolaos Karapanos, Alexandros Filios, Raluca Ada Popa, and Srdjan Capkun. "Verena: End-to-End Integrity Protection for Web Applications". In: *S&P*. IEEE, 2016.

[263] Nikolaos P. Karvelas, Andreas Peter, and Stefan Katzenbeisser. "Using Oblivious RAM in Genomic Studies". In: *DPM CBT*. Springer, 2017.

[264] Akihiro Kato, Michael Scott, Tetsutaro Kobayashi, and Yuto Kawahara. *Barreto-Naehrig Curves*. Tech. rep. draft-kasamatsu-bncurves-02. Internet Engineering Task Force, 2016. URL: `https://datatracker.ietf.org/doc/draft-kasamatsu-bncurves/02/`.

[265] Jonathan Katz, Amit Sahai, and Brent Waters. "Predicate Encryption Supporting Disjunctions, Polynomial Equations, and Inner Products". In: *EUROCRYPT*. Springer, 2008.

[266] Marcel Keller, Emmanuela Orsini, and Peter Scholl. "MASCOT: Faster Malicious Arithmetic Secure Computation with Oblivious Transfer". In: *CCS*. ACM, 2016.

[267] Marcel Keller, Valerio Pastro, and Dragos Rotaru. "Overdrive: Making SPDZ Great Again". In: *EUROCRYPT*. Springer, 2018.

[268] Ben Kepes. *Some scary (for some) statistics around file sharing usage*. `https://www.computerworld.com/article/2991924/some-scary-for-some-statistics-around-file-sharing-usage.html`. 2015.

[269] Keybase.io. *Keybase*. `https://keybase.io/`.

[270] Keyless. *Privacy Enhancing Frictionless Authentication — Keyless*. `https://keyless.io/`. Accessed: June 21, 2023.

[271] Rami Khalil, Alexei Zamyatin, Guillaume Felley, Pedro Moreno-Sanchez, and Arthur Gervais. *Commit-Chains: Secure, Scalable Off-Chain Payments*. Cryptology ePrint Archive, Paper 2018/642. 2018. URL: `https://eprint.iacr.org/2018/642`.

[272] Safwan Mahmud Khan and Kevin W. Hamlen. "AnonymousCloud: A Data Ownership Privacy Provider Framework in Cloud Computing". In: *TrustCom*. IEEE, 2012.

[273] Beom Heyn Kim and David Lie. "Caelus: Verifying the Consistency of Cloud Services with Battery-Powered Devices". In: *S&P*. IEEE, 2015.

[274] Hyung-Sin Kim, Michael P Andersen, Kaifei Chen, Sam Kumar, William J. Zhao, Kevin Ma, and David E. Culler. "System Architecture Directions for Post-SoC/32-bit Networked Sensors". In: *SenSys*. ACM, 2018.

[275] Hyung-Sin Kim, Hosoo Cho, Hongchan Kim, and Saewoong Bahk. "DT-RPL: Diverse bidirectional traffic delivery through RPL routing protocol in low power and lossy networks". In: *Computer Networks* 126 (2017).

[276] Hyung-Sin Kim, Hosoo Cho, Myung-Sup Lee, Jeongyeup Paek, JeongGil Ko, and Saewoong Bahk. "MarketNet: An Asymmetric Transmission Power-based Wireless System for Managing e-Price Tags in Markets". In: *SenSys*. ACM, 2015.

[277] Hyung-Sin Kim, Heesu Im, Myung-Sup Lee, Jeongyeup Paek, and Saewoong Bahk. "A Measurement Study of TCP over RPL in Low-power and Lossy Networks". In: *Journal of Communications and Networks* 17.6 (2015).

[278] Hyung-Sin Kim, Sam Kumar, and David E. Culler. "Thread/OpenThread: A Compromise in Low-Power Wireless Multihop Network Architecture for the Internet of Things". In: *IEEE Communications Magazine* 57.7 (2019).

[279] Sukun Kim, Rodrigo Fonseca, Prabal Dutta, Arsalan Tavakoli, David Culler, Philip Levis, Scott Shenker, and Ion Stoica. "Flush: A Reliable Bulk Transport Protocol for Multihop Wireless Networks". In: *SenSys*. ACM, 2007.

[280] Sukun Kim, Shamim Pakzad, David Culler, James Demmel, Gregory Fenves, Steven Glaser, and Martin Turon. "Health Monitoring of Civil Infrastructures Using Wireless Sensor Networks". In: *IPSN*. ACM/IEEE, 2007.

[281] Taechan Kim and Razvan Barbulescu. "Extended Tower Number Field Sieve: A New Complexity for the Medium Prime Case". In: *CRYPTO*. Springer, 2016.

[282] John Kolb, Moustafa AbdelBaky, Randy H. Katz, and David E. Culler. "Core Concepts, Challenges, and Future Directions in Blockchain: A Centralized Tutorial". In: *ACM Computing Surveys* 53.1 (2020).

[283] Vladimir Kolesnikov, Payman Mohassel, and Mike Rosulek. "FleXOR: Flexible Garbling for XOR Gates that Beats Free-XOR". In: *CRYPTO*. Springer, 2014.

[284] Vladimir Kolesnikov and Thomas Schneider. "Improved Garbled Circuit: Free XOR Gates and Applications". In: *ICALP*. Springer, 2008.

[285] Matthias Kovatsch, Martin Lanter, and Zach Shelby. "Californium: Scalable Cloud Services for the Internet of Things with CoAP". In: *IOT*. IEEE, 2014.

[286] RJ Krawiec, Dan Housman, Mark White, Mariya Filipova, Florian Quarre, Dan Barr, Allen Nesbitt, Kate Fedosova, Jason Killmeyer, Adam Israel, and Lindsay Tsai. *Blockchain: Opportunities for Health Care*. `https://www2.deloitte.com/content/dam/Deloitte/us/Documents/public-sector/us-blockchain-opportunities-for-health-care.pdf`. 2016.

[287] Ben Kreuter, Benjamin Mood, abhi shelat, and Kevin Butler. "PCF: A Portable Circuit Format for Scalable Two-Party Secure Computation". In: *USENIX Security*. USENIX, 2013.

[288] Benjamin Kreuter, abhi shelat, and Chi-hao Shen. "Billion-Gate Secure Computation with Malicious Adversaries". In: *USENIX Security*. USENIX, 2012.

[289] Andrew Krioukov, Gabe Fierro, Nikita Kitaev, and David Culler. "Building Application Stack (BAS)". In: *BuildSys*. ACM, 2012.

[290] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. "OceanStore: An Architecture for Global-Scale Persistent Storage". In: *ASPLOS*. ACM, 2000.

[291] Nishant Kumar, Mayank Rathee, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. "CRYPTFLOW: Secure TensorFlow Inference". In: *S&P*. IEEE, 2020.

[292] Sam Kumar, Michael P Andersen, Hyung-Sin Kim, and David E. Culler. "Demo Abstract: Bringing Full-Scale TCP to Low-Power Networks". In: *SenSys*. ACM, 2018.

[293] Sam Kumar, Michael P Andersen, Hyung-Sin Kim, and David E. Culler. "Performant TCP for Low-Power Wireless Networks". In: *NSDI*. USENIX, 2020.

[294] Sam Kumar, Stuart Cheshire, and others from Apple. *Designing a TCP Stack for OpenThread*. `https://github.com/openthread/openthread/files/6400253/openthread_tcp_design_implementation.pdf`. See also: `https://github.com/openthread/openthread/issues/6456`. Accessed: July 15, 2023. 2021.

[295] Sam Kumar, David E. Culler, and Raluca Ada Popa. "MAGE: Nearly Zero-Cost Virtual Memory for Secure Computation". In: *OSDI*. USENIX, 2021.

[296] Sam Kumar, Yuncong Hu, Michael P Andersen, Raluca Ada Popa, and David E. Culler. "JEDI: Many-to-Many End-to-End Encryption and Key Delegation for IoT". In: *USENIX Security*. USENIX, 2019.

[297] Sam Kumar, Yuncong Hu, Michael P Andersen, Raluca Ada Popa, and David E. Culler. *JEDI: Many-to-Many End-to-End Encryption and Key Delegation for IoT*. 2019. arXiv: `1905.13369 [cs.CR]`.

[298] James F. Kurose and Keith W. Ross. *Computer Networking: A Top-Down Approach*. 6th ed. Pearson, 2013. Chap. 3, pp. 278–279.

[299] Nandakishore Kushalnagar, Gabriel Montenegro, and Christian Schumacher. *IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs): Overview, Assumptions, Problem Statement, and Goals*. RFC 4919. 2007.

[300] Eyal Kushilevitz and Rafail Ostrovsky. "Replication Is Not Needed: Single Database, Computationally-Private Information Retrieval". In: *FOCS*. IEEE, 1997.

[301] Laakeri (`https://cs.stackexchange.com/users/95646/laakeri`). *Is there an algorithm to minimize working set during a topological traversal?* Computer Science Stack Exchange. `https://cs.stackexchange.com/q/120274`. 2020.

[302] Leslie Lamport. "Paxos Made Simple". In: *SIGACT News (Distributed Computing Column)* 32.4 (2001).

[303] Leslie Lamport. "The Part-Time Parliament". In: *TOCS* 16.2 (1998).

[304] Billy Lau, Simon Chung, Chengyu Song, Yeongjin Jang, Wenke Lee, and Alexandra Boldyreva. "Mimesis Aegis: A Mimicry Privacy Shield—A System's Approach to Data Privacy on Public Cloud". In: *USENIX Security*. USENIX, 2014.

[305] Ben Laurie. "Certificate Transparency: Public, verifiable, append-only logs". In: *Queue* 12.8 (2014).

[306] Ryan Lehmkuhl, Pratyush Mishra, Akshayaram Srinivasan, and Raluca Ada Popa. "MUSE: Secure Inference Resilient to Malicious Clients". In: *USENIX Security*. USENIX, 2021.

[307] Robert Lemos. *Home Depot estimates data on 56 million cards stolen by cybercriminals*. `https://arstechnica.com/information-technology/2014/09/home-depot-estimates-data-on-56-million-cards-stolen-by-cybercrimnals/`. Accessed: April 21, 2019.

[308] Philip Levis, Nelson Lee, Matt Welsh, and David Culler. "TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications". In: *SenSys*. ACM, 2003.

[309] Philip Levis, Samuel Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, and David Culler. "TinyOS: An Operating System for Sensor Networks". In: *Ambient Intelligence*. Ed. by Werner Weber, Jan M. Rabaey, and Emile Aarts. Springer, 2005.

[310] Philip Levis, Neil Patel, David Culler, and Scott Shenker. "Trickle: A Self-Regulating Algorithm for Code Propagation and Maintenance in Wireless Sensor Networks". In: *NSDI*. USENIX, 2004.

[311] Amit Levy, James Hong, Laurynas Riliskis, Philip Levis, and Keith Winstein. "Beetle: Flexible Communication for Bluetooth Low Energy". In: *MobiSys*. ACM, 2016.

[312] Henry M. Levy. *Capability-Based Computer Systems*. Digital Press, 1984.

[313] Allison Lewko, Amit Sahai, and Brent Waters. "Revocation Systems with Very Small Private Keys". In: *S&P*. IEEE, 2010.

[314] Cheng Li, Zhenjiang Li, Mo Li, Forrest Meggers, Arno Schlueter, and Hock Beng Lim. "Energy Efficient HVAC System with Distributed Sensing and Control". In: *ICDCS*. IEEE, 2014.

[315] Jinyuan Li, Maxwell Krohn, David Mazières, and Dennis Shasha. "Secure Untrusted Data Repository (SUNDR)". In: *OSDI*. USENIX, 2004.

[316] Yun-Chen Li and Mei-Ling Chiang. "LyraNET: A Zero-Copy TCP/IP Protocol Stack for Embedded Operating Systems". In: *RTCSA*. IEEE, 2005.

[317] Chieh-Jan Mike Liang, Nissanka Bodhi Priyantha, Jie Liu, and Andreas Terzis. "Surviving Wi-Fi Interference in Low Power ZigBee Networks". In: *SenSys*. ACM, 2010.

[318] Benoît Libert and Damien Vergnaud. "Adaptive-ID Secure Revocable Identity-Based Encryption". In: *CT-RSA*. Springer, 2009.

[319] Yehuda Lindell and Benny Pinkas. "An Efficient Protocol for Secure Two-Party Computation in the Presence of Malicious Adversaries". In: *EUROCRYPT*. Springer, 2007.

[320] C. L. Liu and James W. Layland. "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment". In: *J. ACM* 20.1 (1973).

[321] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. "ObliVM: A Programming Framework for Secure Computation". In: *S&P*. IEEE, 2015.

[322] Jian Liu, Mika Juuti, Yao Lu, and N. Asokan. "Oblivious Neural Network Predictions via MiniONN Transformations". In: *CCS*. ACM, 2017.

[323] Weiran Liu, Jianwei Liu, Qianhong Wu, Bo Qin, David Naccache, and Houda Ferradi. *Compact CCA2-secure Hierarchical Identity-Based Broadcast Encryption for Fuzzy-entity Data Sharing*. Cryptology ePrint Archive, Paper 2016/634. 2016. URL: `https://eprint.iacr.org/2016/634`.

[324] *Low Power, 2.4GHz Transceiver for ZigBee, RF4CE, IEEE 802.15.4, 6LoWPAN, and ISM Applications*. AT86RF233. Preliminary Datasheet. Atmel Corporation. 2014.

[325] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation". In: *PLDI*. ACM, 2005.

[326] Chris Maeda and Brian N. Bershad. "Protocol Service Decomposition for High-Performance Networking". In: *SOSP*. ACM, 1993.

[327] Matteo Maffei, Giulio Malavolta, Manuel Reinert, and Dominique Schröder. "Maliciously Secure Multi-Client ORAM". In: *ACNS*. Springer, 2017.

[328] Matteo Maffei, Giulio Malavolta, Manuel Reinert, and Dominique Schröder. "Privacy and Access Control for Outsourced Personal Records". In: *S&P*. IEEE, 2015.

[329] Prince Mahajan, Srinath Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Mike Dahlin, and Michael Walfish. "Depot: Cloud storage with minimal trust". In: *OSDI*. USENIX, 2010.

[330] Alan Mainwaring, Joseph Polastre, Robert Szewczyk, David Culler, and John Anderson. "Wireless Sensor Networks for Habitat Monitoring". In: *WSNA*. ACM, 2002.

[331] Dahlia Malkhi, Michael Merritt, and Ohad Rodeh. "Secure Reliable Multicast Protocols in a WAN". In: *ICDCS*. IEEE, 1997.

[332] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. "Fairplay—A Secure Two-Party Computation System". In: *USENIX Security*. USENIX, 2004.

[333] Dalia Malki and Michael Reiter. "A High-Throughput Secure Reliable Multicast Protocol". In: *CSFW*. IEEE, 1996.

[334] Petros Maniatis and Mary Baker. "Secure History Preservation Through Timeline Entanglement". In: *USENIX Security*. USENIX, 2002.

[335] Hasan Al Maruf and Mosharaf Chowdhury. “Effectively Prefetching Remote Memory with Leap”. In: *ATC*. USENIX, 2020.

[336] Matthew Mathis, Jeffrey Semke, Jamshid Mahdavi, and Teunis Ott. “The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm”. In: *SIGCOMM-CCR* 27.3 (1997).

[337] David Mazières and Dennis Shasha. “Building secure file systems out of Byzantine storage”. In: *PODC*. ACM, 2002.

[338] Drew McDaniel. *Virtual Machines Best Practices: Single VMs, Temporary Storage and Uploaded Disks*. `https://azure.microsoft.com/en-us/blog/virtual-machines-best-practices-single-vms-temporary-storage-and-uploaded-disks/`. Accessed: June 21, 2023. 2014.

[339] Adrian McEwen. *Risking a Compuserve of Things*. `https://mcqn.com/posts/wuthering-bytes-slides-risking-a-compuserve-of-things/`. Accessed: December 8, 2018. 2013.

[340] *Medicalchain - Blockchain for electronic health records*. `https://medicalchain.com`. Accessed: September 12, 2019.

[341] Almir Mehanovic, Thomas Heine Rasmussen, and Mikkel Baun Kjærgaard. “Brume - A Horizontally Scalable and Fault Tolerant Building Operating System”. In: *IoTDI*. IEEE, 2018.

[342] Marcela S. Melara, Aaron Blankstein, Joseph Bonneau, Edward W. Felten, and Michael J. Freedman. “CONIKS: Bringing Key Transparency to End Users”. In: *USENIX Security*. USENIX, 2015.

[343] Adrian Mettler, David Wagner, and Tyler Close. “Joe-E: A Security-Oriented Subset of Java”. In: *NDSS*. Internet Society, 2010.

[344] Microsoft Azure. *Ddv4 and Ddsv4-series*. `https://docs.microsoft.com/en-us/azure/virtual-machines/ddv4-ddsv4-series`. Accessed: June 21, 2023.

[345] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. “DELPHI: A Cryptographic Inference Service for Neural Networks”. In: *USENIX Security*. USENIX, 2020.

[346] Payman Mohassel and Yupeng Zhang. “SecureML: A System for Scalable Privacy-Preserving Machine Learning”. In: *S&P*. IEEE, 2017.

[347] Gabriel Montenegro, Nandakishore Kushalnagar, Jonathan Hui, and David Culler. *Transmission of IPv6 Packets over IEEE 802.15.4 Networks*. RFC 4944. 2007.

[348] Benjamin Mood, Debayan Gupta, Henry Carter, Kevin R. B. Butler, and Patrick Traynor. “Frigate: A Validated, Extensible, and Efficient Compiler and Interpreter for Secure Computation”. In: *EuroS&P*. IEEE, 2016.

[349] Benjamin Mood, Lara Letaw, and Kevin Butler. “Memory-Efficient Garbled Circuit Generation for Mobile Devices”. In: *FC*. Springer, 2012.

[350] Todd C. Mowry, Angela K. Demke, and Orran Krieger. "Automatic Compiler-Inserted I/O Prefetching for Out-of-Core Applications". In: *OSDI*. USENIX, 1996.

[351] *MPC Alliance*. `https://www.mpcalliance.org/`. Accessed: June 30, 2023.

[352] *MQTT*. `http://mqtt.org`. Accessed: January 25, 2018.

[353] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. `https://bitcoin.org/bitcoin.pdf`. 2008.

[354] Dalit Naor, Moni Naor, and Jeff Lotspiech. "Revocation and Tracing Schemes for Stateless Receivers". In: *CRYPTO*. Springer, 2001.

[355] Antonio L. Maia Neto, Artur L. F. Souza, Italo Cunha, Michele Nogueira, Ivan Oliveira Nunes, Leonardo Cotta, Nicolas Gentille, Antonio A. F. Loureiro, Diego F. Aranha, Harsh Kupwade Patil, and Leonardo B. Oliveira. "AoT: Authentication and Access Control for the Entire IoT Device Life-Cycle". In: *SenSys*. ACM, 2016.

[356] Lucien K. L. Ng and Sherman S. M. Chow. "SoK: Cryptographic Neural-Network Computation". In: *S&P*. IEEE, 2023.

[357] Jakob Nielsen. *Nielsen's Law of Internet Bandwidth*. `https://www.nngroup.com/articles/law-of-bandwidth/`. Accessed: May 26, 2020.

[358] Valeria Nikolaenko, Udi Weinsberg, Stratis Ioannidis, Marc Joye, Dan Boneh, and Nina Taft. "Privacy-Preserving Ridge Regression on Hundreds of Millions of Records". In: *S&P*. IEEE, 2013.

[359] *Open mHealth*. `http://www.openmhealth.org/`. Accessed: September 19, 2019.

[360] *OpenThread*. `https://openthread.io`.

[361] Esteban Ordano, Ariel Meilich, Yemel Jardi, and Manuel Araoz. *Decentraland: A blockchain-based virtual world*. `https://decentraland.org/whitepaper.pdf`. 2017.

[362] Fredrik Österlind and Adam Dunkels. "Approaching the Maximum 802.15.4 Multi-hop Throughput". In: *HotEmNets*. ACM, 2008.

[363] Rafail Ostrovsky, Amit Sahai, and Brent Waters. "Attribute-Based Encryption with Non-Monotonic Access Structures". In: *CCS*. ACM, 2007.

[364] Vinicius Pacheco and Ricardo Puttini. "SaaS Anonymous Cloud Service Consumption Structure". In: *ICDCS*. IEEE, 2012.

[365] Jitendra Padhye, Victor Firoiu, Don Towsley, and Jim Kurose. "Modeling TCP Throughput: A Simple Model and its Empirical Validation". In: *SIGCOMM*. ACM, 1998.

[366] Jeongyeup Paek and Ramesh Govindan. "RCRT: Rate-controlled Reliable Transport for Wireless Sensor Networks". In: *SenSys*. ACM, 2007.

[367] Pascal Paillier. "Public-Key Cryptosystems Based on Composite Degree Residuosity Classes". In: *EUROCRYPT*. Springer, 1999.

[368] Qixiang Pang, Vincent W. S. Wong, and Victor C. M. Leung. "Reliable Data Transport and Congestion Control in Wireless Sensor Networks". In: *International Journal of Sensor Networks* 3.1 (2008).

[369] Antonis Papadimitriou, Ranjita Bhagwan, Nishanth Chandran, Ramachandran Ramjee, Andreas Haeberlen, Harmeet Singh, Abhishek Modi, and Saikrishna Badrinarayanan. "Big Data Analytics over Encrypted Datasets with Seabed". In: *OSDI*. USENIX, 2016.

[370] *Particle Mesh*. `https://www.particle.io/mesh`. Accessed: February 2, 2019.

[371] Vern Paxson, Mark Allman, Scott Dawson, Bill Fenner, Jim Griner, Ian Heavens, Kevin Lahey, Jeff Semke, and Bernie Volz. *Known TCP Implementation Problems*. RFC 2525. 1999.

[372] PayPal. *PayPal to Acquire Curv*. `https://newsroom.paypal-corp.com/2021-03-08-PayPal-to-Acquire-Curv`. Accessed: June 21, 2023.

[373] Tony Peng. *Shared Machine Learning: Ant Financial's Solution for Data Privacy*. `https://medium.com/syncedreview/shared-machine-learning-ant-financials-solution-for-data-privacy-8069cffe7bb6`. Accessed: June 21, 2023.

[374] Adrian Perrig, Ran Canetti, J. D. Tygar, and Dawn Song. "Efficient Authentication and Signing of Multicast Streams over Lossy Channels". In: *S&P*. IEEE, 2000.

[375] Adrian Perrig, Robert Szewczyk, Victor Wen, David Culler, and J. D. Tygar. "SPINS: Security Protocols for Sensor Networks". In: *MobiCom*. ACM, 2001.

[376] Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams. "Secure Two-Party Computation Is Practical". In: *ASIACRYPT*. Springer, 2009.

[377] Benny Pinkas, Thomas Schneider, and Michael Zohner. "Faster Private Set Intersection Based on OT Extension". In: *USENIX Security*. USENIX, 2014.

[378] Rishabh Poddar, Sukrit Kalra, Avishay Yanai, Ryan Deng, Raluca Ada Popa, and Joseph M. Hellerstein. "Senate: A Maliciously-Secure MPC Platform for Collaborative Analytics". In: *USENIX Security*. USENIX, 2021.

[379] Joseph Polastre, Jason Hill, and David Culler. "Versatile Low Power Media Access for Wireless Sensor Networks". In: *SenSys*. ACM, 2004.

[380] Joseph Polastre, Robert Szewczyk, and David Culler. "Telos: Enabling Ultra-Low Power Wireless Research". In: *IPSN*. ACM/IEEE, 2005.

[381] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. "CryptDB: Protecting Confidentiality with Encrypted Query Processing". In: *SOSP*. ACM, 2011.

[382] Raluca Ada Popa, Emily Stark, Jonas Helfer, Steven Valdez, Nickolai Zeldovich, M. Frans Kaashoek, and Hari Balakrishnan. "Building Web Applications on Top of Encrypted Data Using Mylar". In: *USENIX Security*. USENIX, 2014.

[383] PreVeil Inc. *PreVeil: End-to-end encryption for everyone*. `preveil.com`.

[384] Privly. *Privly*. priv.ly.

[385] The FreeBSD Project. *FreeBSD 10.3*. https://www.freebsd.org/releases/10.3R/announce.html. 2016.

[386] Lucy Qin. *Deploying MPC for Social Good*. Real World Crypto. https://youtu.be/5pkDq4sRWyQ?t=2090. 2019.

[387] Michael O. Rabin. *How to Exchange Secrets with Oblivious Transfer*. Tech. rep. TR-81. Harvard University, 1981.

[388] Md. Abdur Rahman, Abdulmotaleb El Saddik, and Wail Gueaieb. "Wireless Sensor Network Transport Layer: State of the Art". In: *Sensors: Advancements in Modeling, Design Issues, Fabrication and Practical Applications*. Ed. by Subhas Chandra Mukhopadhyay and Yueh-Min Huang. Springer, 2008.

[389] Costin Raiciu, Christoph Paasch, Sebastien Barre, Alan Ford, Michio Honda, Fabien Duchene, Olivier Bonaventure, and Mark Handley. "How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP". In: *NSDI*. USENIX, 2012.

[390] Anantha Ramaiah, Randall R. Stewart, and Mitesh Dalal. *Improving TCP's Robustness to Blind In-Window Attacks*. RFC 5961. 2010.

[391] B. Randell. "A Note on Storage Fragmentation and Program Segmentation". In: *CACM* 12.7 (1969).

[392] Deevashwer Rathee, Mayank Rathee, Nishant Kumar, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. "CrypTFlow2: Practical 2-Party Secure Inference". In: *CCS*. ACM, 2020.

[393] A. J. Dinusha Rathnayaka and Vidyasagar M. Potdar. "Wireless Sensor Network transport protocol: A critical review". In: *Journal of Network and Computer Applications* 36.1 (2013).

[394] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. "A Scalable Content-Addressable Network". In: *SIGCOMM*. ACM, 2001.

[395] Oded Regev. "On Lattices, Learning with Errors, Random Linear Codes, and Cryptography". In: *STOC*. ACM, 2005.

[396] James Reyes. *Building the next generation of digital advertising with MPC*. Real World Crypto. https://youtu.be/6Gb0xO8csVU?t=2533. 2022.

[397] M. Sadegh Riazi, Mohammad Samragh, Hao Chen, Kim Laine, Kristin Lauter, and Farinaz Koushanfar. "Xonn: XNOR-based Oblivious Deep Neural Network Inference". In: *USENIX Security*. USENIX, 2019.

[398] Mike Rosulek. *A Brief History of Practical Garbled Circuit Optimizations*. https://simons.berkeley.edu/talks/mike-rosulek-2015-06-09, https://www.youtube.com/watch?v=FTxh908u9y8. Accessed: April 27, 2020. 2015.

[399] Bita Darvish Rouhani, M. Sadegh Riazi, and Farinaz Koushanfar. "DeepSecure: Scalable Provably-Secure Deep Learning". In: *DAC*. ACM, 2018.

[400] Antony Rowstron and Peter Druschel. "Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems". In: *Middleware*. Springer, 2001.

[401] *rsablind*. `https://github.com/cryptoballot/rsablind`.

[402] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. "AIFM: High-Performance, Application-Integrated Far Memory". In: *OSDI*. USENIX, 2020.

[403] Amit Sahai and Hakan A. Seyalioglu. "Worry-Free Encryption: Functional Encryption with Public Keys". In: *CCS*. ACM, 2010.

[404] Amit Sahai and Brent Waters. "Fuzzy Identity-Based Encryption". In: *EUROCRYPT*. Springer, 2005.

[405] Cetin Sahin, Victor Zakhary, Amr El Abbadi, Huijia Lin, and Stefano Tessaro. "TaoStore: Overcoming Asynchronicity in Oblivious Data Storage". In: *S&P*. IEEE, 2016.

[406] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Nathan Manohar, Nicholas Genise, Srinivas Devadas, Karim Eldefrawy, Chris Peikert, and Daniel Sanchez. "CraterLake: A Hardware Accelerator for Efficient Unbounded Computation on Encrypted Data". In: *ISCA*. ACM, 2022.

[407] Yogesh Sankarasubramaniam, Özgür B. Akan, and Ian F. Akyildiz. "ESRT: Event-to-Sink Reliable Transport in Wireless Sensor Networks". In: *MobiHoc*. ACM, 2003.

[408] Danilo F. S. Santos, Hyggo O. Almeida, and Angelo Perkusich. "A personal connected health system for the Internet of Things based on the Constrained Application Protocol". In: *Computers and Electrical Engineering* 44 (2015).

[409] Thomas Schmid, Roy Shea, Mani B. Srivastava, and Prabal Dutta. "Disentangling Wireless Sensing from Mesh Networking". In: *HotEmNets*. ACM, 2010.

[410] *Microsoft SEAL (release 3.6)*. `https://github.com/Microsoft/SEAL`. Microsoft Research, Redmond, WA. 2020.

[411] Tara Seals. *17% of Workers Fall for Social Engineering Attacks*. `https://www.infosecurity-magazine.com/news/17-of-workers-fall-for-social/`. 2018.

[412] *Secret Double Octopus — Passwordless High Assurance Authentication*. `https://doubleoctopus.com`. Accessed: April 21, 2019.

[413] Klara Seitz, Sebastian Serth, Konrad-Felix Krentz, and Christoph Meinel. "Demo Abstract: Enabling En-Route Filtering for End-to-End Encrypted CoAP Messages". In: *SenSys*. ACM, 2017.

[414] Jeffrey Semke, Jamshid Mahdavi, and Matthew Mathis. "Automatic TCP Buffer Tuning". In: *SIGCOMM*. ACM, 1998.

[415] Jae Hong Seo and Keita Emura. "Efficient Delegation of Key Generation and Revocation Functionalities in Identity-Based Encryption". In: *CT-RSA*. Springer, 2013.

[416] Jae Hong Seo and Keita Emura. “Revocable Hierarchical Identity-Based Encryption: History-Free Update, Security Against Insiders, and Short Ciphertexts”. In: *CT-RSA*. Springer, 2015.

[417] Jae Hong Seo and Keita Emura. “Revocable Identity-Based Encryption Revisited: Security Model and Construction”. In: *PKC*. Springer, 2013.

[418] Hossein Shafagh, Lukas Burkhalter, Sylvia Ratnasamy, and Anwar Hithnawi. “Droplet: Decentralized Authorization and Access Control for Encrypted Data Streams”. In: *USENIX Security*. USENIX, 2020.

[419] Hossein Shafagh, Anwar Hithnawi, Lukas Burkhalter, Pascal Fischli, and Simon Duquennoy. “Secure Sharing of Partially Homomorphic Encrypted IoT Data”. In: *SenSys*. ACM, 2017.

[420] Hossein Shafagh, Anwar Hithnawi, Andreas Dröscher, Simon Duquennoy, and Wen Hu. “Talos: Encrypted Query Processing for the Internet of Things”. In: *SenSys*. ACM, 2015.

[421] Adi Shamir. “How to Share a Secret”. In: *CACM* 22.11 (1979).

[422] Adi Shamir. “Identity-Based Cryptosystems and Signature Schemes”. In: *CRYPTO*. Springer, 1984.

[423] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. “EROS: a fast capability system”. In: *SOSP*. ACM, 1999.

[424] Janae Sharp. *Will Healthcare See Ethical Patient Data Exchange?* `https://www.idigitalhealth.com/news/healthcare-ethical-patient-data-exchange-cms-rule`. Accessed: September 12, 2019.

[425] Zach Shelby. *Java speaks CoAP*. `https://community.arm.com/iot/b/blog/posts/java-speaks-coap`. Accessed: September 9, 2018. 2015.

[426] Zach Shelby, Klaus Hartke, and Carsten Bormann. *The Constrained Application Protocol (CoAP)*. RFC 7252. 2014.

[427] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. “Oblivious RAM with $O((\log N)^3)$ Worst-Case Cost”. In: *ASIACRYPT*. Springer, 2011.

[428] Victor Shoup. “Practical Threshold Signatures”. In: *EUROCRYPT*. Springer, 2000.

[429] Alexander Shraer, Christian Cachin, Asaf Cidon, Idit Keidar, Yan Michalevsky, and Dani Shaket. “Venus: Verification for Untrusted Cloud Storage”. In: *CCSW*. ACM, 2010.

[430] *Sia*. `https://sia.tech`. Accessed: April 16, 2019.

[431] Nigel Smart. *Multi-Party Computation: A Cryptographic Marvel in Search of Its Commercial Sweet Spot*. `https://www.eeweb.com/multi-party-computation-a-cryptographic-marvel-in-search-of-its-commercial-sweet-spot/`. Accessed: June 30, 2023. 2022.

[432] Alex C. Snoeren and Hari Balakrishnan. “An End-to-End Approach to Host Mobility”. In: *MobiCom*. ACM, 2000.

[433] *Software Configuration Guide, Cisco IOS Release 15.2(5)EX (Catalyst Digital Building Series Switches)*. https://www.cisco.com/c/en/us/td/docs/switches/lan/catalyst_digital_building_series_switches/software/15-2_5_ex/configuration_guide/b_1525ex_consolidated_cdb_cg/b_1525ex_consolidated_cdb_cg_chapter_010001.html. Accessed: June 21, 2023. Cisco, 2018.

[434] Solace. *Advanced Event Broker. An event mesh for connected enterprises — Solace*. https://solace.com. Accessed: January 17, 2018.

[435] Zhenyu Song, Daniel S. Berger, Kai Li, and Wyatt Lloyd. "Learning Relaxed Belady for Content Distribution Network Caching". In: *NSDI*. USENIX, 2020.

[436] Ebrahim M. Songhori, Siam U. Hussain, Ahmad-Reza Sadeghi, Thomas Schneider, and Farinaz Koushanfar. "TinyGarble: Highly Compressed and Scalable Sequential Garbled Circuits". In: *S&P*. IEEE, 2015.

[437] Mudhakar Srivatsa and Mike Hicks. "Deanonymizing Mobility Traces: Using Social Networks as a Side-Channel". In: *CCS*. ACM, 2012.

[438] Fred Stann and John Heidemann. "RMST: Reliable Data Transport in Sensor Networks". In: *SNPA*. IEEE, 2003.

[439] Thanos Stathopoulos, Lewis Girod, John Heidemann, and Deborah Estrin. *Mote Herding for Tiered Wireless Sensor Networks*. Tech. rep. 58. Center for Embedded Networked Computing, University of California, Los Angeles, 2005. URL: http://www.isi.edu/%7ejohnh/PAPERS/Stathopoulos05a.html.

[440] Emil Stefanov and Elaine Shi. "ObliviStore: High Performance Oblivious Cloud Storage". In: *S&P*. IEEE, 2013.

[441] *Decentralized Cloud Storage — Storj*. https://storj.io. Accessed: April 16, 2019.

[442] Swarm Team. *Swarm: Storage and Communication Infrastructure for a Self-Sovereign Digital Society*. https://www.ethswarm.org/swarm-whitepaper.pdf. 2021.

[443] Robert Szewczyk, Joseph Polastre, Alan Mainwaring, and David Culler. "Lessons from a Sensor Network Expedition". In: *EWSN*. Springer, 2004.

[444] Ankur Taly and Asim Shankar. "Distributed Authorization in Vanadium". In: *FOSAD VIII*. Springer, 2016.

[445] Sijun Tan, Brian Knott, Yuan Tian, and David J. Wu. "CryptGPU: Fast Privacy-Preserving Machine Learning on the GPU". In: *S&P*. IEEE, 2021.

[446] Muhammad Adnan Tariq, Boris Koldehofe, and Kurt Rothermel. "Securing Broker-Less Publish/Subscribe Systems Using Identity-Based Encryption". In: *TPDS* 25.2 (2014).

[447] Moti N. Thadani and Yousef A. Khalidi. *An Efficient Zero-Copy I/O Framework for UNIX*. Tech. rep. SMLI TR-95-39. Sun Microsystems Laboratories, Inc., 1995.

[448] *Thread Benefits*. https://www.threadgroup.org/What-is-Thread/Thread-Benefits. Accessed: June 21, 2023.

[449] *Thread Group*. `https://www.threadgroup.org/thread-group`. Accessed: June 21, 2023.

[450] *Thread Group*. `https://threadgroup.org`.

[451] Alin Tomescu and Srinivas Devadas. "Catena: Efficient Non-equivocation via Bitcoin". In: *S&P*. IEEE, 2017.

[452] Omri Traub, Glenn Holloway, and Michael D. Smith. "Quality and Speed in Linear-scan Register Allocation". In: *PLDI*. ACM, 1998.

[453] Tresorit. *End-to-end encrypted cloud storage*. `tresorit.com`.

[454] Viktor Trón, Aron Fischer, and Nick Johnson. *smash-proof: Auditable storage for Swarm secured by masked audit secret hash*. 2016.

[455] Nirvan Tyagi, Yossi Gilad, Derek Leung, Matei Zaharia, and Nickolai Zeldovich. "Stadium: A Distributed Metadata-Private Messaging System". In: *SOSP*. ACM, 2017.

[456] Wen-Guey Tzeng. "A Time-Bound Cryptographic Key Assignment Scheme for Access Control in a Hierarchy". In: *TKDE* 14.1 (2002).

[457] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. "Benchmarking, Analysis, and Optimization of Serverless Function Snapshots". In: *ASPLOS*. ACM, 2021.

[458] JP Vasseur. *Terms Used in Routing for Low-Power and Lossy Networks*. RFC 7102. 2014.

[459] Alexander Viand, Patrick Jattke, and Anwar Hithnawi. "SoK: Fully Homomorphic Encryption Compilers". In: *S&P*. IEEE, 2021.

[460] Berta Carballido Villaverde, Dirk Pesch, Rodolfo De Paz Alberola, Szymon Fedor, and Menouer Boubekeur. "Constrained Application Protocol for Low Power Embedded Networks: A Survey". In: *IMIS*. IEEE, 2012.

[461] Virtru. *Virtru: Email Encryption and Data Protection Solutions*. `www.virtru.com`.

[462] Nikolaj Volgushev, Malte Schwarzkopf, Ben Getchell, Mayank Varia, Andrei Lapets, and Azer Bestavros. "Conclave: secure multi-party computation on big data". In: *EuroSys*. ACM, 2019.

[463] *VOLTTRON*. `https://volttron.org/`. Accessed: January 23, 2019.

[464] Amanda Walker, Sarvar Patel, and Moti Yung. *Helping organizations do more without collecting more data*. Google Security Blog. `https://security.googleblog.com/2019/06/helping-organizations-do-more-without-collecting-more-data.html`. 2019.

[465] Chieh-Yih Wan, Andrew T. Campbell, and Lakshman Krishnamurthy. "PSFQ: A Reliable Transport Protocol for Wireless Sensor Networks". In: *WSNA*. ACM, 2002.

[466] Chieh-Yih Wan, Shane B. Eisenman, and Andrew T. Campbell. "CODA: Congestion Detection and Avoidance in Sensor Networks". In: *SenSys*. ACM, 2003.

[467] Chonggang Wang, Kazem Sohraby, Yueming Hu, Bo Li, and Weiwen Tang. "Issues of Transport Control Protocols for Wireless Sensor Networks". In: *ICCCAS*. IEEE, 2005.

[468] Frank Wang, James Mickens, Nickolai Zeldovich, and Vinod Vaikuntanathan. "Sieve: Cryptographically Enforced Access Control for User Data in Untrusted Clouds". In: *NSDI*. USENIX, 2016.

[469] Frank Wang, Catherine Yun, Shafi Goldwasser, Vinod Vaikuntanathan, and Matei Zaharia. "Splinter: Practical Private Queries on Public Data". In: *NSDI*. USENIX, 2017.

[470] Guojun Wang, Qin Liu, and Jie Wu. "Hierarchical Attribute-Based Encryption for Fine-Grained Access Control in Cloud Storage Services". In: *CCS*. ACM, 2010.

[471] Guojun Wang, Qin Liu, Jie Wu, and Minyi Guo. "Hierarchical attribute-based encryption and scalable user revocation for sharing data in cloud servers". In: *Computers & Security* 30.5 (2011).

[472] Ke Coby Wang and Michael K. Reiter. "How to End Password Reuse on the Web". In: *NDSS*. Internet Society, 2019.

[473] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. *EMP-toolkit: Efficient MultiParty computation toolkit*. `https://github.com/emp-toolkit`. 2016.

[474] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. "Authenticated Garbling and Efficient Maliciously Secure Two-Party Computation". In: *CCS*. ACM, 2017.

[475] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. "Global-Scale Secure Multiparty Computation". In: *CCS*. ACM, 2017.

[476] Xiao Shaun Wang, T-H. Hubert Chan, and Elaine Shi. "Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound". In: *CCS*. ACM, 2015.

[477] Xinlei Wang, Jianqing Zhang, Eve M. Schooler, and Mihaela Ion. "Performance Evaluation of Attribute-Based Encryption: Toward Data Privacy in the IoT". In: *ICC*. IEEE, 2014.

[478] Yohei Watanabe, Keita Emura, and Jae Hong Seo. "New Revocable IBE in Prime-Order Groups: Adaptively Secure, Decryption Key Exposure Resistant, and with Short Public Parameters". In: *CT-RSA*. Springer, 2017.

[479] Jean-Luc Watson, Sameer Wagh, and Raluca Ada Popa. "Piranha: A GPU Platform for Secure Computation". In: *USENIX Security*. USENIX, 2022.

[480] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. "Ceph: A Scalable, High-Performance Distributed File System". In: *OSDI*. USENIX, 2006.

[481] WhatsApp. *WhatsApp's Privacy Notice*. `www.whatsapp.com/legal/?doc=privacy-policy`. 2012.

[482] Ben Whittle. *Storing Documents on the Blockchain: Why, How, and Where*. `https://coincentral.com/storing-documents-on-the-blockchain-why-how-and-where/`. Accessed: June 23, 2023.

[483] Christian Wimmer and Hanspeter Mössenböck. “Optimized Interval Splitting in a Linear Scan Register Allocator”. In: *VEE*. ACM, 2005.

[484] Phil Windley. *The CompuServe of Things*. `http://www.windley.com/archives/2014/04/the_compuserve_of_things.shtml`. Accessed: December 8, 2018. 2014.

[485] Keith Winstein and Hari Balakrishnan. “Mosh: An Interactive Remote Shell for Mobile Clients”. In: *ATC*. USENIX, 2012.

[486] Keith Winstein, Anirudh Sivaraman, and Hari Balakrishnan. “Stochastic Forecasts Achieve High Throughput and Low Delay over Cellular Networks”. In: *NSDI*. USENIX, 2013.

[487] Tim Winter, Pascal Thubert, Anders Brandt, Jonathan Hui, Richard Kelsey, Philip Levis, Kris Pister, Rene Struik, JP Vasseur, and Roger Alexander. *RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks*. RFC 6550. 2012.

[488] Alec Woo and David E. Culler. “A Transmission Control Scheme for Media Access in Sensor Networks”. In: *MobiCom*. ACM, 2001.

[489] Gary R. Wright and W. Richard Stevens. *TCP/IP Illustrated*. Vol. 2. Addison-Wesley Publishing Company, 1995. Chap. 2.

[490] David J. Wu, Ankur Taly, Asim Shankar, and Dan Boneh. “Privacy, Discovery, and Authentication for the Internet of Things”. In: *ESORICS*. Springer, 2016.

[491] Howard Wu, Wenting Zheng, Alessandro Chiesa, Raluca Ada Popa, and Ion Stoica. “DIZK: A Distributed Zero Knowledge Proof System”. In: *USENIX Security*. USENIX, 2018.

[492] Ning Xu, Sumit Rangwala, Krishna Kant Chintalapudi, Deepak Ganesan, Alan Broad, Ramesh Govindan, and Deborah Estrin. “A Wireless Sensor Network For Structural Monitoring”. In: *SenSys*. ACM, 2004.

[493] Vijay Kumar Yadav, Nitish Andola, Shekhar Verma, and S. Venkatesan. “A Survey of Oblivious Transfer Protocol”. In: *ACM Computing Surveys* 54.10 (2022).

[494] Sophia Yakoubov. *A Gentle Introduction to Yao’s Garbled Circuits*. `http://web.mit.edu/sonka89/www/papers/2017ygc.pdf`. Accessed: April 27, 2020. 2017.

[495] Andrew C. Yao. “Protocols for Secure Computations”. In: *FOCS*. IEEE, 1982.

[496] Andrew Chi-Chih Yao. “How to Generate and Exchange Secrets”. In: *FOCS*. IEEE, 1986.

[497] Danfeng Yao, Nelly Fazio, Yevgeniy Dodis, and Anna Lysyanskaya. “ID-based Encryption for Complex Hierarchies with Applications to Forward Security and Broadcast Encryption”. In: *CCS*. ACM, 2004.

[498] Wei Ye, John Heidemann, and Deborah Estrin. “An energy-efficient MAC protocol for wireless sensor networks”. In: *INFOCOM*. IEEE, 2002.

[499] Wei Ye, John Heidemann, and Deborah Estrin. “Medium Access Control with Coordinated Adaptive Sleeping for Wireless Sensor Networks”. In: *IEEE/ACM Transactions on Networking* 12.3 (2004).

[500] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. "HotStuff: BFT Consensus with Linearity and Responsiveness". In: *PODC*. ACM, 2019.

[501] Shucheng Yu, Cong Wang, Kui Ren, and Wenjing Lou. "Achieving Secure, Scalable, and Fine-grained Data Access Control in Cloud Computing". In: *INFOCOM*. IEEE, 2010.

[502] Thomas Zachariah, Noah Klugman, Bradford Campbell, Joshua Adkins, Neal Jackson, and Prabal Dutta. "The Internet of Things Has a Gateway Problem". In: *HotMobile*. ACM, 2015.

[503] Samee Zahur and David Evans. *Obliv-C: A Language for Extensible Data-Oblivious Computation*. Cryptology ePrint Archive, Paper 2015/1153. 2015. URL: `https://eprint.iacr.org/2015/1153`.

[504] Samee Zahur, Mike Rosulek, and David Evans. "Two Halves Make a Whole: Reducing Data Transfer in Garbled Circuits Using Half Gates". In: *EUROCRYPT*. Springer, 2015.

[505] Saman Zarandioon, Danfeng (Daphne) Yao, and Vinod Ganapathy. "K2C: Cryptographic Cloud Storage with Lazy Revocation and Anonymous Access". In: *SecureComm*. Springer, 2011.

[506] Zcash. *Zcash: All coins are created equal*. `http://z.cash/`.

[507] *ZeroMQ*. `http://zeromq.org`. Accessed: January 29, 2019.

[508] Kim Zetter. *'Google' Hackers Had Ability to Alter Source Code*. `https://www.wired.com/2010/03/source-code-hacks/`. Accessed: April 21, 2019.

[509] Kim Zetter. *An Unprecedented Look at Stuxnet, the World's First Digital Weapon*. `https://www.wired.com/2014/11/countdown-to-zero-day-stuxnet/`. Accessed: April 21, 2019.

[510] Hongwei Zhang, Anish Arora, Young-ri Choi, and Mohamed G. Gouda. "Reliable Bursty Convergecast in Wireless Sensor Networks". In: *MobiHoc*. ACM, 2005.

[511] Irene Zhang, Alex Garthwaite, Yury Baskakov, and Kenneth C. Barr. "Fast Restore of Checkpointed Memory using Working Set Estimation". In: *VEE*. ACM, 2011.

[512] Lixia Zhang. "Why TCP Timers Don't Work Well". In: *SIGCOMM*. ACM, 1986.

[513] Tiancong Zheng, Ahmed Ayadi, and Xiaoran Jiang. "TCP over 6LoWPAN for Industrial Applications: An Experimental Study". In: *NTMS*. IEEE, 2011.

[514] Wenting Zheng, Ryan Deng, Weikeng Chen, Raluca Ada Popa, Aurojit Panda, and Ion Stoica. "Cerebro: A Platform for Multi-Party Cryptographic Collaborative Learning". In: *USENIX Security*. USENIX, 2021.

[515] Wenting Zheng, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. "Helen: Maliciously Secure Coopetitive Learning for Linear Models". In: *S&P*. IEEE, 2019.

[516] Ruiyu Zhu, Darion Cassel, Amr Sabry, and Yan Huang. "NANOPI: Extreme-Scale Actively-Secure Multi-Party Computation". In: *CCS*. ACM, 2018.

[517] *Zigbee Gateway*. https://www.zigbee.org/zigbee-for-developers/zigbee-gateway/. Accessed: February 13, 2019.

Appendix A

Ghostor's Security Guarantees

This appendix formally describes Ghostor's privacy and security guarantees.

A.1 Ghostor's Privacy Guarantee

In this appendix, we use the simulation paradigm of Secure Multi-Party Computation (SMPC) to define Ghostor's privacy guarantee. We begin in Appendix A.1.1 by providing an overview of our definition and proof sketch, along with an explanation of how our simulation-based definition matches the one in Section 6.3.3.

A.1.1 Overview

We formally define Ghostor's anonymity by specifying an *ideal world*. We provided a definition in Section 6.3.3, but we consider it to be informal because it does not clearly state what the adversary learns if some users are compromised/malicious. The ideal world is specified such that it is easy to reason about what information the adversary learns; what the adversary learns in the ideal world is our definition of what an anonymous object sharing system leaks to an adversary (i.e., what *anonymity* does not hide). In contrast, we refer to a setup running the actual Ghostor protocol as the *real world*. Below, we refer to the formalized Ghostor protocol as $\pi_{\text{Ghostor}}^{\text{Payment}}$; as we explain in Appendix A.1.1.2, this includes some minor differences from Section 6.7. The "Payment" in the notation indicates that we are working in the $\mathcal{F}_{\text{Payment}}$-hybrid model, which we also explain in Appendix A.1.1.2.

In the real world, clients interact directly with the server $\mathcal{A}$. We denote the server as $\mathcal{A}$ because it is controlled by an adversary. The party P embodies the honest clients. When a client issues an API call, P interacts with $\mathcal{A}$ according to Ghostor's protocol to perform the API call on behalf of the client. $\mathcal{A}$ cannot directly inspect the clients' state or secret information, but it may attempt to infer it through the messages it receives from P as it interacts with P to serve each request.

In the ideal world, clients interact with an incorruptible trusted party $\mathcal{F}$ called an *ideal functionality*. On each API call issued by a client, $\mathcal{F}$ provides another party, $\mathcal{S}$, with a well-defined subset

of information in the API call. The subset of information that $\mathcal{F}$ gives to $\mathcal{S}$ defines what information Ghostor leaks to the adversary, and provides a clear definition of what anonymity means in our setting. After receiving this information from $\mathcal{F}$, $\mathcal{S}$ interacts with $\mathcal{A}$ to perform the corresponding API call. Obviously, $\mathcal{S}$ cannot perform the operation exactly as the client specified because it does not know the entire API call, only the *subset* of the API call that $\mathcal{F}$ gives it. The challenge is for $\mathcal{S}$ to *simulate* the operation such that what $\mathcal{A}$ sees is cryptographically indistinguishable from what if would see if the same API call were made in the real world. The existence of $\mathcal{S}$ that can properly simulate $\pi^{\text{Payment}}_{\text{Ghostor}}$ toward $\mathcal{A}$ would show that Ghostor reveals no more to $\mathcal{A}$ than what $\mathcal{F}$ gives $\mathcal{S}$ on each API call.

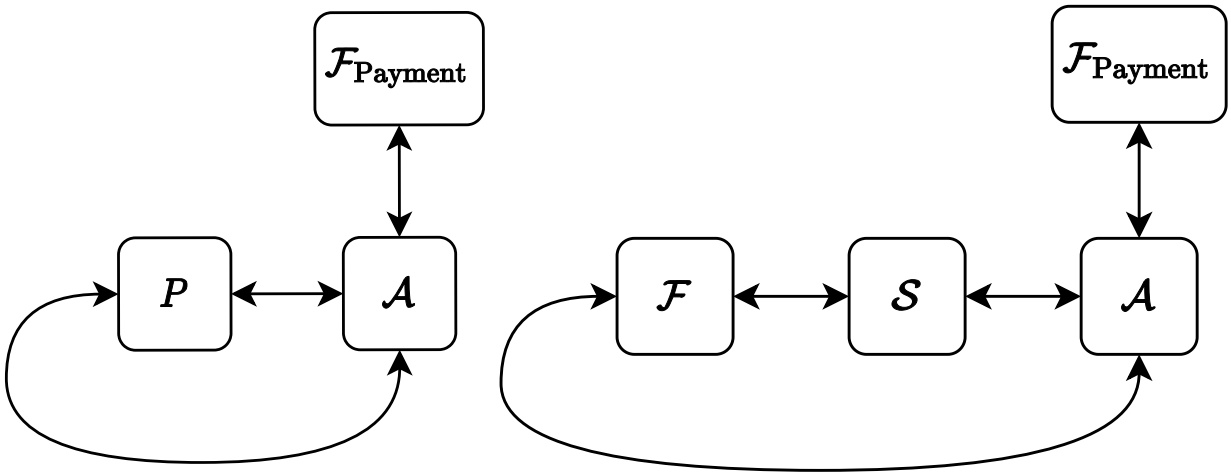
We allow $\mathcal{A}$ to adaptively choose the API calls issued by *honest* users, by instructing the client P which API calls to make in the real world and specifying these API calls to $\mathcal{F}$ in the ideal world (e.g., instructing a particular user to `GET` a particular object). Once each API call is completed, $\mathcal{A}$ receives the return value of the API call (e.g., the object contents that are the result of a `GET`) from P in the real world and $\mathcal{F}$ in the ideal world. To capture that Ghostor does not directly leak this information to the adversary, our ideal world has $\mathcal{A}$ specify API calls directly to P in the real world and $\mathcal{F}$ in the ideal world and receive the return values directly from P or $\mathcal{F}$, bypassing the simulator $\mathcal{S}$ to make API calls and receive responses. Thus, $\mathcal{A}$ is *external* to $\mathcal{S}$. Having $\mathcal{A}$ adaptively choose what API calls honest users issue and see their return values strengthens our definition; it shows that our anonymity guarantees still hold if the adversary happens to observe the output of some clients' operations, through some side channel outside of the Ghostor system. Although $\mathcal{A}$ chooses the API calls issued by honest users, $\mathcal{A}$ cannot see the internal state of honest users. In particular, $\mathcal{A}$ cannot access honest users' secret keys.

For malicious and compromised users, $\mathcal{A}$ *internally* interacts with the server on their behalf, without interacting with P in the real world or $\mathcal{F}$ in the ideal world. This is necessary because a compromised user may collude with the server $\mathcal{A}$ and interact with storage in a way that is not captured by any API call.

Figure A.1 provides a high-level summary and comparison of the real world and ideal world.

A.1.1.1 Map to Definition of Anonymity in Section 6.3.3

In Section 6.3.3, we explained Ghostor's privacy guarantee in terms of a leakage function. The leakage function in Section 6.3.3 is largely the same as the information that $\mathcal{F}$ gives to $\mathcal{S}$ on each API call (Appendix A.1.3.2). There are a few minor differences, which we now explain. Timing information is not included in Appendix A.1.3.2 because the model we use in our cryptographic formalization does not have a notion of time. That said, the order in which the requests are processed is given to $\mathcal{S}$; it is implicit in the order in which $\mathcal{F}$ sends messages to it. Finally, although not explicit in Appendix A.1.3.2, $\mathcal{S}$ can infer how many round trips are performed between the client and server in processing each operation. Because we do not model concurrent operations and there is no client-side caching of data (Section 6.4.4), the adversary can infer how many round trips are required from the client-server protocol (Section 6.7), so this does not reveal any meaningful information.



(a) Real World.

(b) Ideal World.

Figure A.1: Overview of Real World and Ideal World.

Our definition of anonymity matches the everyday use of the word "anonymity" because $\mathcal{S}$ does not receive any user-specific information for operations issued by honest users on objects that no compromised user is authorized to access. Furthermore, $\mathcal{S}$ does not see the membership of the system (public keys of users) or even know how many users exist in the system, apart from corrupt/malicious users.

A.1.1.2 Limitations of our Formalization

Although our cryptographic formalization is useful to prove Ghostor's anonymity, there are some aspects of Ghostor that it does not model. We use the notation $\pi^{\text{Payment}}_{\text{Ghostor}}$ to describe our formalization of Ghostor's protocol from Section 6.7, including these differences. First, we do not directly model the anonymous payment (e.g., Zcash) aspect of Ghostor. Instead, we assume the existence of an ideal functionality for Zcash, $\mathcal{F}_{\text{Payment}}$, that can be queried to validate payment (i.e., learn how much was paid and when). To denote that we are working in the $\mathcal{F}_{\text{Payment}}$-hybrid model, we denote the Ghostor protocol used in the real world as $\pi^{\text{Payment}}_{\text{Ghostor}}$. Second, we do not directly model network information (e.g., IP addresses) leaked to the server when clients connect, because this is hidden by the use of an anonymity network like Tor (Section 6.9). Third, whereas the Ghostor system allows operations to be processed concurrently (i.e., round trips of different operations may be interleaved), our formalization $\pi^{\text{Payment}}_{\text{Ghostor}}$ assumes that the Ghostor server processes each operation one at a time (because P will only answer one request at a time). Fourth, we do not fully model Ghostor's integrity mechanisms, such as epochs, checkpoints, or the return value of `obtain_digests`. This is because Ghostor's integrity guarantees can only be verified at the end of the epoch; $\mathcal{A}$ may commit arbitrary integrity violations during an epoch. Therefore, it is not meaningful to provide integrity guarantees for individual Ghostor operations. We analyze Ghostor's integrity in

Appendix A.2.

Users may also be malicious (i.e., controlled by the adversary). In our formalization, the adversary may compromise users, but we restrict the adversary to doing so *statically*. This means that the adversary compromises users at the time of their creation. Specifically, the adversary may not create an honest user, have that honest user perform operations, and then later compromise that user.

A.1.2 Real World

In the real world, the client P interacts directly with the server $\mathcal{A}$ according to $\pi^{\text{Payment}}_{\text{Ghostor}}$.

To request P to perform an API call, $\mathcal{A}$ sends an Initiate message to P. In the Initiate message, $\mathcal{A}$ can request P to perform the following API calls:

- `create_user`() $\rightarrow$ userID
- `learn_pk`(userID) $\rightarrow$ pk or $\perp$
- `create_object`(userID, ACL, contents, token) $\rightarrow$ (objectID, digest) or $\perp$
- `set_acl`(userID, objectID, ACL, contents) $\rightarrow$ digest or $\perp$
- `PUT`(userID, objectID, contents) $\rightarrow$ digest or $\perp$
- `GET`(userID, objectID) $\rightarrow$ (contents, digest) or $\perp$
- `obtain_tokens`(paymentID) $\rightarrow \{\text{token}_i\}_i$ or $\perp$
- `obtain_digests`(objectID) $\rightarrow$ data or $\perp$

Here, $\perp$ is a symbol representing failure. This occurs if P detects an issue and returns an error. For example, this happens if the server $\mathcal{A}$ denies service (e.g., server provides malformed messages or object data that is not signed according to $\pi^{\text{Payment}}_{\text{Ghostor}}$).

In the above API, ACL is represented as a list of tuples, where each tuple contains a user and a set of permission bits for that user. If padding is desired, additional tuples for the owner can be added. The user can be identified by either a userID or a pk; allowing the user to be represented by the pk allows $\mathcal{A}$ to add malicious users it created internally (who do not have userIDs) to the ACL of an object.

The `learn_pk` operation is not part of the Ghostor API (Section 6.2). We use it to model the *out-of-band* exchange of public keys in Ghostor. The server $\mathcal{A}$ can use this operation to learn an honest user's public key, which it can then use to (internally) interact with a malicious user to add that honest user to an object's ACL.

In Ghostor, objects are identified by their PVK, so the objectID in Ghostor is the bits of the PVK. We use objectID so that our definition is not tied to PVKs, which are specific to Ghostor's design.

For payment, we model Zcash as an ideal functionality $\mathcal{F}_{\text{Payment}}$, which allows $\mathcal{A}$ to make a payment and obtain a paymentID, or validate a paymentID and learn how much was paid and when.

The Initiate message contains an opcode, specifying which operation to perform, and the arguments to that operation. After receiving an Initiate message, P performs the operation. For a `create_user` operation, P generates a keypair for a user and a uniformly random userID for that user, locally maintains the mapping from userID to keypair, and returns userID back to $\mathcal{A}$. On a `learn_pk` operation, P gives $\mathcal{A}$ public key pk for the provided userID. For all other operations, P

interacts with $\mathcal{A}$ according to the protocol in Section 6.7, and gives the final result of the API call (e.g., plaintext object contents in the case of `GET`) back to $\mathcal{A}$.

A.1.3 Ideal World

We define an ideal functionality $\mathcal{F}$ for an anonymous object sharing system in the simulation paradigm, which captures Ghostor's privacy guarantee. Our notation and setup are as follows. $\mathcal{A}$ may send an Initiate message to $\mathcal{F}$, specifying an API call to be made on behalf of an honest user. To perform the operation, $\mathcal{F}$ interacts with the simulator $\mathcal{S}$, which simulates $\pi^{\text{Payment}}_{\text{Ghostor}}$ toward the server $\mathcal{A}$ based on the subset of each API call that $\mathcal{F}$ provides to Ghostor. At the end of the interaction, $\mathcal{F}$ obtains a return value, which it gives to $\mathcal{A}$. As in the real world, there exists a party $\mathcal{F}_{\text{Payment}}$ in the ideal world with which $\mathcal{A}$ can interact.

A.1.3.1 State Maintained by $\mathcal{F}$

As $\mathcal{F}$ sees all `create_user` and `learn_pk` operations and their return values, it maintains a list of all honest users $\mathcal{A}$ is aware of and their public keys. We call this structure the *user table*. Based on the user table, $\mathcal{F}$ can tell whether a public key corresponds to an honest user; if it does not, it assumes the public key corresponds to a malicious user that $\mathcal{A}$ crafted internally.

Similarly, as $\mathcal{F}$ sees all `create_object` and `set_acl` operations and their return values, it maintains a list of all objects created by honest users and all ACLs that have been attempted to be set for each object. We call this structure the *permission table*. In the discussion below, we say that an object is *tainted* if either (1) it was not created by an honest user (i.e., if it was crafted by $\mathcal{A}$), or (2) it was created by an honest user but a malicious user has been on the ACL of the object, either currently or in the past. Crucially, $\mathcal{F}$ identifies if an object is tainted or not based on its permission table.

In general, $\mathcal{F}$ does not provide plaintext data to $\mathcal{S}$ for objects that are not tainted, but must give plaintext data in return values given to $\mathcal{A}$. When $\mathcal{F}$ receives plaintext object data m from $\mathcal{A}$, it generates a fresh identifier called a contentID for that data. contentID can, for example, be a number chosen sequentially. The tuple $(m, \text{contentID})$ is stored by $\mathcal{F}$, allowing $\mathcal{F}$ to later recover m from the contentID. We call the set of tuples of the form $(m, \text{contentID})$ the *content table*.

Finally, $\mathcal{F}$ maintains a mapping from tokens obtained by honest users to an identifier sampled uniformly at random. We call this mapping the *token table* and refer to a token's random identifier as its *anonym*. Each anonym is, by definition, unlinkable with the particular token it corresponds to.

A.1.3.2 Description of $\mathcal{F}$

When $\mathcal{F}$ receives an Initiate message from $\mathcal{A}$, it performs some processing and then reveals to $\mathcal{S}$ the opcode (except for `create_user`) and the following information:

- For `create_user`(), $\mathcal{F}$ generates a uniformly random userID for the user and gives it to $\mathcal{A}$. For this operation *only*, $\mathcal{F}$ gives nothing to $\mathcal{S}$, not even the opcode. $\mathcal{F}$ updates its user table with the new userID.
- For `learn_pk`(userID), $\mathcal{F}$ looks in its user table to check if userID corresponds to a user who was created with `create_user`. If not, $\mathcal{F}$ returns $\perp$. If so, $\mathcal{F}$ gives the message `learn_pk`(userID) to $\mathcal{S}$. $\mathcal{S}$ responds with a pk, which $\mathcal{F}$ gives to $\mathcal{A}$.
- For `create_object`(userID, ACL, contents, token), $\mathcal{F}$ scans the ACL of the new object and identifies non-honest users using its user table. Then, it generates ACL$'$, a randomly shuffled list consisting of the records from ACL that correspond to non-honest users. It also computes c, the size (number of records) of the ACL. $\mathcal{F}$ looks up token in its token table to obtain a, the corresponding anonym. If the object is tainted (which occurs if ACL contains at least one malicious user), then $\mathcal{F}$ gives the message `create_object`(ACL$'$, c, contents, a) to $\mathcal{S}$. Otherwise, $\mathcal{F}$ generates a fresh contentID and adds the entry (contents, contentID) to its content table, and then $\mathcal{F}$ gives the message `create_object`(ACL$'$, c, contentID, ℓ, a) to $\mathcal{S}$, where ℓ is the length of contents. $\mathcal{S}$ responds with the return value, and $\mathcal{F}$ checks if the operation was successful (if the return value is not $\perp$). If so, $\mathcal{F}$ adds a new entry to its permission table with objectID, the creator userID, and the ACL of the object. Finally, $\mathcal{F}$ gives the return value to $\mathcal{A}$.
- For `set_acl`(userID, objectID, ACL, contents), $\mathcal{F}$ scans the ACL and identifies non-honest users using its user table. Then, it generates ACL$'$, a randomly shuffled list consisting of the records from ACL that correspond to non-honest users. It also computes c, the size (number of records) of the ACL. If the object was created by an honest user, $\mathcal{F}$ computes a bit b indicating whether userID corresponds to a user authorized to perform a `set_acl` operation on this object (which is only true if that user created the object); it can find this information by checking its permission table. If the object was not created by an honest user, then $\mathcal{F}$ gives the message `set_acl`(userID, objectID, ACL$'$, c, contents) to $\mathcal{S}$. If the object was created by an honest user but is tainted, then $\mathcal{F}$ gives the message `set_acl`(objectID, ACL$'$, c, contents, b) to $\mathcal{S}$. If the object was created by an honest user and is not tainted, then $\mathcal{F}$ generates a fresh contentID and adds the entry (contents, contentID) to its content table, and then $\mathcal{F}$ gives the message `set_acl`(objectID, ACL$'$, c, contentID, ℓ, b) to $\mathcal{S}$, where ℓ is the length of contents. $\mathcal{S}$ responds with the return value and $\mathcal{F}$ gives it to $\mathcal{A}$. $\mathcal{F}$ adds ACL to its permission table as one of the ACLs for the object corresponding to objectID.
- For `PUT`(userID, objectID, contents), $\mathcal{F}$ checks if the object corresponding to objectID was created by an honest user by checking its permission table. If so, it computes a bit vector $\vec{p}$, with one bit per current and prior ACL for the object, recording if it grants the user corresponding to userID permission to perform a `PUT`. If the object was not created by an honest user, then $\mathcal{F}$ gives the message `PUT`(userID, objectID, contents) to $\mathcal{S}$. If the object was created by an honest user but is tainted, then $\mathcal{F}$ gives the message `PUT`(objectID, contents, $\vec{p}$) to $\mathcal{S}$. If the object was created by an honest user but is not tainted, then $\mathcal{F}$ generates a fresh contentID and adds the entry (contents, contentID) to its content table, and then $\mathcal{F}$ gives the message `PUT`(objectID, contentID, ℓ, $\vec{p}$) to $\mathcal{S}$, where ℓ is the length of contents. $\mathcal{S}$ responds with the return value and $\mathcal{F}$ gives it to $\mathcal{A}$.
- For `GET`(userID, objectID), $\mathcal{F}$ checks if the object corresponding to objectID was created by an

honest user by checking its permission table. If so, it computes a bit vector $\vec{p}$, with one bit per current and prior ACL for the object, recording if it grants the user corresponding to userID permission to perform a GET. If the object was not created by an honest user, then $\mathcal{F}$ gives the message GET(userID, objectID) to $\mathcal{S}$. If the object was created by an honest user, then $\mathcal{F}$ gives the message GET(objectID, $\vec{p}$) to $\mathcal{S}$. $\mathcal{S}$ responds with either $\perp$ or with a two-element tuple where the second element is the digest and the first element is either plaintext object contents or a contentID. If the return value contains a contentID, it is translated to plaintext using the content table, and the resulting return value is given to $\mathcal{A}$.

- For obtain_tokens(paymentID), $\mathcal{F}$ gives the message obtain_tokens(paymentID) to $\mathcal{S}$. $\mathcal{S}$ responds with the return value, and $\mathcal{F}$ gives it to $\mathcal{A}$. If tokens are returned, $\mathcal{F}$ adds entries to its token table for those tokens with fresh anonyms generated uniformly at random.
- For obtain_digests(objectID), $\mathcal{F}$ gives the message obtain_digests(objectID) to $\mathcal{S}$. $\mathcal{S}$ responds with the return value, and $\mathcal{F}$ gives it to $\mathcal{A}$.

A.1.4 Security Definition

Now that we have defined the Real and Ideal Worlds, we are ready to precisely state what it means for Ghostor to be anonymous. We denote the security parameter as κ.

Definition 3 (Anonymous Data-Sharing System)**.** *Let π be the protocol for a data-sharing system (i.e., it provides clients with the API given in Appendix A.1.2). For an adversary $\mathcal{A}$ that outputs a single bit, let* $\text{REAL}_{\pi,\mathcal{A}}(1^\kappa)$ *be the random variable denoting $\mathcal{A}$'s output when interacting with the real world (Appendix A.1.2). For a simulator $\mathcal{S}$ and an adversary $\mathcal{A}$ that outputs a single bit, let* $\text{IDEAL}_{\mathcal{S},\mathcal{A}}(1^\kappa)$ *be the random variable denoting $\mathcal{A}$'s output when interacting with the ideal world (Appendix A.1.3).*

We say that π is anonymous *if there exists a non-uniform algorithm $\mathcal{S}$ probabilistic polynomial-time in κ such that, for every non-uniform algorithm $\mathcal{A}$ probabilistic polynomial-time in κ that outputs a single bit, the probability ensemble of* $\text{REAL}_{\pi,\mathcal{A}}(1^\kappa)$ *over κ is computationally indistinguishable from the probability ensemble of* $\text{IDEAL}_{\mathcal{S},\mathcal{A}}(1^\kappa)$ *over κ.*

That is,

$$\exists \mathcal{S}\ \forall \mathcal{A}\ \left\{\text{REAL}_{\pi,\mathcal{A}}(1^\kappa)\right\}_\kappa \stackrel{c}{\equiv} \left\{\text{IDEAL}_{\mathcal{S},\mathcal{A}}(1^\kappa)\right\}_\kappa$$

Based on this definition, we state the following security guarantee for Ghostor.

Theorem 4 (Privacy in Ghostor)**.** *Suppose that in Ghostor, the data encryption scheme is semantically secure [191], the encryption scheme for key list entries in the object header is semantically secure [191] and key-private [44], payment tokens are blind [105, 95], $\mathcal{F}_{Payment}$ is an ideal functionality for Zcash, and signatures are existentially unforgeable [191]. Then $\pi_{Ghostor}^{Payment}$ is anonymous as defined in Definition 3.*

A.1.5 Proof of Ghostor's Privacy

We prove Theorem 4 by constructing a simulator $\mathcal{S}$ for Ghostor that, given the information provided by $\mathcal{F}$ on each API call, interacts with $\mathcal{A}$ in a way $\mathcal{A}$ cannot distinguish the real world from the ideal world. For readability, we use the language "$\mathcal{A}$ cannot distinguish the real world from the ideal world" throughout this section to mean the cryptographic equivalence stated in Definition 3 applied to $\pi^{\text{Payment}}_{\text{Ghostor}}$, namely $\{\text{REAL}_{\pi^{\text{Payment}}_{\text{Ghostor}},\mathcal{A}}(1^\kappa)\}_\kappa \stackrel{c}{\equiv} \{\text{IDEAL}_{\mathcal{S},\mathcal{A}}(1^\kappa)\}_\kappa$.

$\mathcal{S}$ interacts with $\mathcal{A}$ over multiple round trips just like a Ghostor client would. $\mathcal{F}$ is designed not to give $\mathcal{S}$ any user identities for untainted objects, yet $\mathcal{S}$ needs to interact with $\mathcal{A}$ as *some* user. The key idea is that $\mathcal{S}$ creates a single *dummy* user keypair, and performs interaction with $\mathcal{A}$ on behalf of honest users on untainted objects using that keypair. Ghostor is designed such that the server cannot distinguish this from a separate keypair being consistently used for each honest user, for untainted objects.

A.1.5.1 State Maintained by $\mathcal{S}$

To process a `learn_pk` API call, $\mathcal{S}$ generates a keypair $(\mathsf{pk},\mathsf{sk})$ for a particular userID. $\mathcal{S}$ must remember the association between the user and the keypair, so it stores the association in a *keypair table*. This structure is similar to $\mathcal{F}$'s user table, but it has two major differences: (1) it includes the pair $(\mathsf{pk},\mathsf{sk})$ for each user instead of just the public key pk, and (2) it only includes users for which a `learn_pk` operation has been issued.

On certain API calls (e.g., `PUT`), $\mathcal{S}$ receives a $\mathsf{contentID}$ from $\mathcal{F}$, but needs to give $\mathcal{A}$ a message that is indistinguishable from what it would receive from an actual client. Therefore, $\mathcal{S}$ generates a fake ciphertext f_ℓ—an encryption of a "zero string" of the same length ℓ as the plaintext message—and interacts with $\mathcal{A}$ using this fake ciphertext. When it does this, $\mathcal{S}$ locally stores a tuple of the form $(\mathsf{contentID}, f_\ell)$ so that it can later associate that particular fake ciphertext f_ℓ with the original content ID (e.g., when performing a `GET` operation on the object). We call the set of tuples of the form $(\mathsf{contentID}, f_\ell)$ the *ciphertext table*.

To process a `create_object` API call, $\mathcal{S}$ receives the anonym a corresponding to the token that was used, but not the token itself. While tokens are indistinguishable as long as they are used only once, $\mathcal{A}$ may reuse tokens, in which case $\mathcal{S}$ will receive the same anonym more than once. To ensure that the same token is reused when interacting with $\mathcal{A}$, just as $\mathcal{A}$ would perceive in the real world, $\mathcal{S}$ maintains an *anonym table* mapping anonyms to tokens provided by $\mathcal{A}$. This mapping is different from $\mathcal{F}$'s token table; the same token may correspond to different anonyms in the two tables. Furthermore, $\mathcal{S}$ maintains a *free token pool* consisting of tokens issued by $\mathcal{A}$ that do not correspond to an entry in the anonym table.

Because $\mathcal{S}$ processes `create_object` and `set_acl` operations, it knows the keys (Table 6.2) associated with each ACL written to each object created by an honest user. $\mathcal{S}$ maintains a *header table* that maps each object header (consisting of fake and real ciphertexts) that $\mathcal{S}$ sends to the server to the associated keys for interacting with that object.

Finally, $\mathcal{S}$ generates the *dummy keypair* $(\mathsf{pk}_0,\mathsf{sk}_0)$ which is used as a stand-in for honest users that the adversary $\mathcal{A}$ cannot identify.

A.1.5.2 Description of $\mathcal{S}$

We now explain how $\mathcal{S}$ interacts with $\mathcal{A}$ upon receiving information from $\mathcal{S}$. In the itemization below, the parameters to each API call are the data that $\mathcal{F}$ gives to $\mathcal{S}$, not the data that $\mathcal{A}$ gives to $\mathcal{F}$. For all operations where $\mathcal{S}$ interacts with $\mathcal{A}$, if the server $\mathcal{A}$ deviates from the protocol in a way that is immediately detectable by the client (e.g., a signature does not check) then $\mathcal{S}$ chooses $\perp$ as the return value and gives it to $\mathcal{F}$.

- For `learn_pk`(userID), $\mathcal{S}$ checks if userID already has an entry in its keypair table. If so, it looks up the public key pk for userID and gives the return value pk to $\mathcal{S}$. If not, it generates a new user keypair $(\mathsf{pk}, \mathsf{sk})$, adds the entry $(\mathsf{userID}, (\mathsf{pk}, \mathsf{sk}))$ to its keypair table, and gives the return value pk to $\mathcal{S}$.
- For `create_object`$(\mathsf{ACL}', c, \mathsf{contents}, a)$, $\mathcal{S}$ checks its anonym table to see if a already has an entry. If so, it retrieves the corresponding token. If not, it chooses a token uniformly at random from the free token pool, removes it from the pool, and adds the entry (a, token) to its anonym table. $\mathcal{S}$ samples fresh keys for the new object header h. To construct h's key list, it (1) initializes the key list with entries based on the records in ACL', (2) pads h's key list to a length of c by encrypting zero strings of the same length as a normal key list entry using the dummy public key pk_0, and (3) randomly shuffles the key list entries in h. After constructing h, it adds an entry to its header table containing h and the freshly sampled keys. Then, it interacts with $\mathcal{A}$ to perform a `create_object` operation, following the real-world protocol with the following changes: (1) token is used for payment, and (2) h is used as the object header. The contents of the object are initialized to contents. $\mathcal{S}$ updates its header table by adding an entry with the object header and the object keypairs generated to perform this operation. If the operation completes successfully, $\mathcal{S}$ returns $(\mathsf{PVK}, \mathsf{digest})$ to $\mathcal{F}$, where PVK is the permission verifying key that $\mathcal{S}$ generated in the course of interacting with $\mathcal{A}$ to create the object and digest is the digest returned by the server.
- For `create_object`$(\mathsf{ACL}', c, \mathsf{contentID}, \ell, a)$, $\mathcal{S}$ checks its anonym table to see if a already has an entry. If so, it retrieves the corresponding token. If not, it chooses a token uniformly at random from the free token pool, removes it from the pool, and adds the entry (a, token) to its anonym table. $\mathcal{S}$ samples fresh keys for the new object header h. To construct h's key list, it (1) initializes the key list with entries based on the records in ACL', (2) pads h's key list to a length of c by encrypting zero strings of the same length as a normal key list entry using the dummy public key pk_0, and (3) randomly shuffles the key list entries in h. After constructing h, it adds an entry to its header table containing h and the freshly sampled keys. $\mathcal{S}$ generates a zero string of length ℓ, encrypts it with the freshly sampled OSK to obtain f_ℓ, and adds the entry $(\mathsf{contentID}, f_\ell)$ to its ciphertext table. Then, it interacts with $\mathcal{A}$ to perform a `create_object` operation, following the real-world protocol with the following changes: (1) token is used for payment, (2) h is used as the object header, and (3) f_ℓ is used as the ciphertext of the object contents. $\mathcal{S}$ updates its header table by adding an entry with the object header and the object keypairs generated to perform this operation. If the operation completes successfully, $\mathcal{S}$ returns $(\mathsf{PVK}, \mathsf{digest})$ to $\mathcal{F}$, where PVK is the permission verifying key that $\mathcal{S}$ generated in the course of interacting with $\mathcal{A}$ to create the object and digest is the digest returned by the server.
- For `set_acl`$(\mathsf{userID}, \mathsf{objectID}, \mathsf{ACL}', c, \mathsf{contents})$, $\mathcal{S}$ samples fresh keys for the new object header

h. To construct h's key list, it (1) initializes the key list with entries based on the records in ACL', (2) pads h's key list to a length of c by encrypting zero strings of the same length as a normal key list entry using the dummy public key pk_0, and (3) randomly shuffles the key list entries in h. After constructing h, it adds an entry to its header table containing h and the freshly sampled keys. Then it looks up the keypair $(\mathsf{pk}, \mathsf{sk})$ corresponding to userID in its keypair table. If that userID is not found in the keypair table, then $\mathcal{S}$ interacts with $\mathcal{A}$ but aborts after downloading the object header from $\mathcal{A}$, as would a normal Ghostor client upon failure to find a suitable entry in the object header. If userID was found in the keypair table, then $\mathcal{S}$ interacts with $\mathcal{A}$ to perform a `set_acl` operation on the object corresponding to $\mathsf{objectID}$, following the real-world protocol with the following changes: (1) h is used as the new object header, and (2) given the existing header h' provided by $\mathcal{A}$, $\mathcal{S}$ first checks its header table to obtain the object keys (including PSK), and if that fails, tries to use the keypair $(\mathsf{pk}, \mathsf{sk})$ to obtain those keys from h' (or abort, as before, if no suitable entry in the object header is found). If the operation completes successfully, $\mathcal{S}$ obtains the return value digest from $\mathcal{A}$ and gives it to $\mathcal{F}$.

- For $\texttt{set_acl}(\mathsf{objectID}, \mathsf{ACL}', c, \mathsf{contents}, b)$, $\mathcal{S}$ samples fresh keys for the new object header h. To construct h's key list, it (1) initializes the key list with entries based on the records in ACL', (2) pads h's key list to a length of c by encrypting zero strings of the same length as a normal key list entry using the dummy public key pk_0, and (3) randomly shuffles the key list entries in h. After constructing h, it adds an entry to its header table containing h and the freshly sampled keys. If b indicates that the user is not authorized to perform this operation, then $\mathcal{S}$ interacts with $\mathcal{A}$ but aborts after downloading the object header from $\mathcal{A}$, as would a normal Ghostor client upon failure to find a suitable entry in the object header. If b indicates that the user is authorized to perform this operation, then $\mathcal{S}$ interacts with $\mathcal{A}$ to perform a `set_acl` operation on the object corresponding to $\mathsf{objectID}$, following the real-world protocol with the following changes: (1) h is used as the new object header, and (2) after obtaining the previous header from $\mathcal{A}$, $\mathcal{S}$ obtains PSK by looking up the header provided by $\mathcal{A}$ in the header table. If the operation completes successfully, $\mathcal{S}$ obtains the return value digest from $\mathcal{A}$ and gives it to $\mathcal{F}$.
- For $\texttt{set_acl}(\mathsf{objectID}, \mathsf{ACL}', c, \mathsf{contentID}, \ell, b)$, $\mathcal{S}$ samples fresh keys for the new object header h. To construct h's key list, it (1) initializes the key list with entries based on the records in ACL', (2) pads h's key list to a length of c by encrypting zero strings of the same length as a normal key list entry using the dummy public key pk_0, and (3) randomly shuffles the key list entries in h. After constructing h, it adds an entry to its header table containing h and the freshly sampled keys. $\mathcal{S}$ generates a zero string of length ℓ, encrypts it with the freshly sampled OSK to obtain f_ℓ, and adds the entry $(\mathsf{contentID}, f_\ell)$ to its ciphertext table. If b indicates that the user is not authorized to perform this operation, then $\mathcal{S}$ interacts with $\mathcal{A}$ but aborts after downloading the object header from $\mathcal{A}$, as would a normal Ghostor client upon failure to find a suitable entry in the object header. If b indicates that the user is authorized to perform this operation, then $\mathcal{S}$ interacts with $\mathcal{A}$ to perform a `set_acl` operation on the object corresponding to $\mathsf{objectID}$, following the real-world protocol with the following changes: (1) h is used as the new object header, (2) after obtaining the previous header from $\mathcal{A}$, $\mathcal{S}$ obtains PSK by looking up the header provided by $\mathcal{A}$ in the header table, and (3) f_ℓ is used as the ciphertext of the object contents. If the operation completes successfully, $\mathcal{S}$ obtains the return value digest from $\mathcal{A}$ and gives it to $\mathcal{F}$.

- For $\texttt{PUT}(\mathsf{userID}, \mathsf{objectID}, \mathsf{contents})$, $\mathcal{S}$ looks up the keypair $(\mathsf{pk}, \mathsf{sk})$ corresponding to userID in its keypair table. If that userID is not found in the keypair table, then $\mathcal{S}$ interacts with $\mathcal{A}$ but aborts after downloading the object header from $\mathcal{A}$, as would a normal Ghostor client upon failure to find a suitable entry in the object header. If userID was found in the keypair table, then $\mathcal{S}$ interacts with $\mathcal{A}$ to perform a $\texttt{PUT}$ operation on the object corresponding to $\mathsf{objectID}$, following the real-world protocol with the following change: given the existing header h' provided by $\mathcal{A}$, $\mathcal{S}$ first checks its header table to obtain the object keys (including WSK and OSK), and if that fails, tries to use the keypair $(\mathsf{pk}, \mathsf{sk})$ to obtain those keys from h' (or abort, as before, if no suitable entry in the object header is found). The object contents are set to $\mathsf{contents}$. If the operation completes successfully, $\mathcal{S}$ obtains the return value digest from $\mathcal{A}$ and gives it to $\mathcal{F}$.
- For $\texttt{PUT}(\mathsf{objectID}, \mathsf{contents}, \vec{p})$, $\mathcal{S}$ interacts with $\mathcal{A}$ to perform a $\texttt{PUT}$ operation on the object corresponding to $\mathsf{objectID}$, following the real-world protocol with the following change. After the existing object header is provided by $\mathcal{A}$, $\mathcal{S}$ checks $\vec{p}$ to see if the user has permission to perform a $\texttt{PUT}$ according to the existing header provided by $\mathcal{A}$. If not, $\mathcal{S}$ aborts the operation, as would a normal Ghostor client upon failure to find a suitable entry in the object header. If so, $\mathcal{S}$ obtains the object keys, including WSK and OSK, by looking up the existing header provided by $\mathcal{A}$ in the header table and then completes the operation, setting the object contents to $\mathsf{contents}$. If the operation completes successfully, $\mathcal{S}$ obtains the return value digest from $\mathcal{A}$ and gives it to $\mathcal{F}$.
- For $\texttt{PUT}(\mathsf{objectID}, \mathsf{contentID}, \ell, \vec{p})$, $\mathcal{S}$ interacts with $\mathcal{A}$ to perform a $\texttt{PUT}$ operation on the object corresponding to $\mathsf{objectID}$, following the real-world protocol with the following change. After the existing object header is provided by $\mathcal{A}$, $\mathcal{S}$ checks $\vec{p}$ to see if the user has permission to perform a $\texttt{PUT}$ according to the existing header provided by $\mathcal{A}$. If not, $\mathcal{S}$ aborts the operation, as would a normal Ghostor client upon failure to find a suitable entry in the object header. If so, $\mathcal{S}$ obtains the object keys, including WSK and OSK, by looking up the existing header provided by $\mathcal{A}$ in the header table. $\mathcal{S}$ generates a zero string of length ℓ, encrypts it with the freshly sampled OSK to obtain f_ℓ, and adds the entry $(\mathsf{contentID}, f_\ell)$ to its ciphertext table. Then $\mathcal{S}$ completes the operation, using f_ℓ as the new object content ciphertext. If the operation completes successfully, $\mathcal{S}$ obtains the return value digest from $\mathcal{A}$ and gives it to $\mathcal{F}$.
- For $\texttt{GET}(\mathsf{userID}, \mathsf{objectID})$, $\mathcal{S}$ looks up the keypair $(\mathsf{pk}, \mathsf{sk})$ corresponding to userID in its keypair table. If that userID is not found in the keypair table, then $\mathcal{S}$ interacts with $\mathcal{A}$ but aborts after downloading the object header from $\mathcal{A}$, as would a normal Ghostor client upon failure to find a suitable entry in the object header. If userID was found in the keypair table, then $\mathcal{S}$ interacts with $\mathcal{A}$ to perform a $\texttt{GET}$ operation on the object corresponding to $\mathsf{objectID}$, following the real-world protocol with the following change: given the existing header h' provided by $\mathcal{A}$, $\mathcal{S}$ first checks its header table to obtain the object keys (including RSK and OSK), and if that fails, tries to use the keypair $(\mathsf{pk}, \mathsf{sk})$ to obtain those keys from h' (or abort, as before, if no suitable entry in the object header is found). If the operation completes successfully, $\mathcal{S}$ obtains the return value $(\mathsf{contents}, \mathsf{digest})$ from $\mathcal{A}$ and gives it to $\mathcal{F}$.
- For $\texttt{GET}(\mathsf{objectID}, \vec{p})$, $\mathcal{S}$ interacts with $\mathcal{A}$ to perform a $\texttt{GET}$ operation on the object corresponding to $\mathsf{objectID}$, following the real-world protocol with the following changes: (1) After the existing object header is provided by $\mathcal{A}$, $\mathcal{S}$ checks $\vec{p}$ to see if the user has permission to perform a $\texttt{GET}$

according to the existing header provided by $\mathcal{A}$. If not, $\mathcal{S}$ aborts the operation, as would a normal Ghostor client upon failure to find a suitable entry in the object header. If so, $\mathcal{S}$ obtains the object keys, including RSK and OSK, by looking up the existing header provided by $\mathcal{A}$ in the header table. (2) Once $\mathcal{S}$ obtains the ciphertext c of the object contents and digest, $\mathcal{S}$ looks in its ciphertext table to obtain the corresponding contentID, and gives the return value (contentID, digest) to $\mathcal{F}$. If $\mathcal{S}$ fails to find a matching entry in the ciphertext table (which happens if c is not a fake ciphertext), then $\mathcal{S}$ decrypts c using OSK to obtain m, and gives the return value (m, digest) to $\mathcal{F}$.

- For `obtain_tokens`(paymentID), $\mathcal{S}$ interacts with $\mathcal{A}$ to perform an `obtain_tokens` operation, providing paymentID as input. $\mathcal{S}$ obtains the return value as a result of interacting with $\mathcal{A}$. If it is not $\perp$, then $\mathcal{S}$ adds the tokens to its free token pool. $\mathcal{S}$ gives the return value to $\mathcal{F}$.
- For `obtain_digests`(objectID), $\mathcal{S}$ interacts with $\mathcal{A}$ to perform an `obtain_digests` operation, providing objectID as input. $\mathcal{S}$ obtains the return value as a result of interacting with $\mathcal{A}$ and gives it to $\mathcal{F}$.

A.1.5.3 Remarks

On `GET` and `PUT` operations, $\mathcal{F}$ reveals to $\mathcal{S}$ a bit vector $\vec{p}$ indicating whether the user is authorized according to each ACL, past and present, of the object. But $\mathcal{S}$ only uses one bit from this vector, namely the one corresponding to the header that the server uses in the protocol. We can close this gap by generalizing our formulation of the ideal world to allow $\mathcal{F}$ to participate in the individual round trips between $\mathcal{S}$ and $\mathcal{A}$. Instead of providing $\mathcal{S}$ with $\vec{p}$ up front, $\mathcal{F}$ can wait until $\mathcal{S}$ receives a header from $\mathcal{A}$, and then only reveal one bit to $\mathcal{S}$ indicating if the user is authorized according to the ACL corresponding to the particular header that $\mathcal{S}$ received. We decided not to do this in our formulation for the sake of simplicity.

Additionally, if contentIDs are allocated sequentially, based on the API calls that $\mathcal{F}$ has forwarded to $\mathcal{S}$, then $\mathcal{F}$ can avoid giving the contentID to $\mathcal{S}$. This is because $\mathcal{S}$ can also keep track of how many operations it has completed, and calculate the contentID based on this count to match the one that $\mathcal{F}$ would have given it. Again, we did not do this in our formulation for the sake of simplicity.

A.1.5.4 Proof Sketch of Indistinguishability

Now, we complete the proof of Theorem 4 by showing that, for the simulator $\mathcal{S}$ described above, no adversary $\mathcal{A}$ can distinguish the real world from the ideal world.

Proof. We will use a sequence of seven hybrid setups to show that no $\mathcal{A}$ can distinguish interacting with the ideal world from interacting with the real world. $\mathcal{H}_0$ is equivalent to the real-world setup, and $\mathcal{H}_6$ is equivalent to the ideal-world setup; below we show that, for all i, $\mathcal{A}$ cannot distinguish interacting with $\mathcal{H}_i$ from interacting with $\mathcal{H}_{i+1}$ with non-negligible probability (i.e., the probability ensemble of $\mathcal{A}$'s output when interacting with $\mathcal{H}_i$ is computationally indistinguishable from the probability ensemble of $\mathcal{A}$'s output when interacting with $\mathcal{H}_{i+1}$). In a true hybrid argument, only

one operation can be modified at a time; our hybrids in the proof sketch below should be interpreted as key stages rather than individual hybrids.

Hybrid $\mathcal{H}_0$. This is exactly the real-world setup in Appendix A.1.2.

Hybrid $\mathcal{H}_1$. This is the same as $\mathcal{H}_0$, except that we rename P to $\mathcal{S}$. $\mathcal{S}$ is responsible for running the real-world protocol to interact with Ghostor based on the *full* information provided in the API calls. $\mathcal{A}$ interacts with $\mathcal{F}_{\text{Payment}}$ as before.

The messages observed by $\mathcal{A}$ are exactly as before, so $\mathcal{A}$ cannot distinguish $\mathcal{H}_0$ from $\mathcal{H}_1$.

Hybrid $\mathcal{H}_2$. This is the same as $\mathcal{H}_1$, except that we now introduce the ideal functionality $\mathcal{F}$. $\mathcal{F}$, in this hybrid, just relays messages back and forth between the adversary $\mathcal{A}$ and the simulator $\mathcal{S}$.

Again, the messages observed by $\mathcal{A}$ are distributed exactly as before, so $\mathcal{A}$ cannot distinguish $\mathcal{H}_1$ from $\mathcal{H}_2$.

Hybrid $\mathcal{H}_3$. This is the same as $\mathcal{H}_2$, except for the following differences: (1) $\mathcal{S}$ maintains the keypair table and header table, (2) $\mathcal{F}$ maintains the user table and permission table and gives b or $\vec{p}$ to $\mathcal{S}$ instead of userID for objects created by honest users, (3) `create_user` and `learn_pk` operations are processed as in the ideal world, and (4) $\mathcal{S}$ uses the dummy keypair $(\mathsf{pk}_0, \mathsf{sk}_0)$ to interact on behalf of honest users with objects created by honest users.

Although $\mathcal{S}$ now relies on lookups in the header table based on headers provided by $\mathcal{A}$ to interact with objects created by honest users, the lookups are guaranteed to succeed because the object header is signed with PSK, which $\mathcal{A}$ does not have for objects created by honest users. Thus, if the adversary attempts to produce a novel header for which the lookup would fail, for an object created by an honest user, it would have to forge a signature, which it cannot do except with negligible probability due to the *existential unforgeability* of the signature scheme. Recall that, if $\mathcal{A}$ were to produce a header that is not properly signed, $\mathcal{S}$ would abort the operation, as would a real-world client, and return $\perp$ to $\mathcal{F}$.

From $\mathcal{A}$'s perspective, all messages it sees are identically distributed with $\mathcal{H}_2$, except that for objects created by honest users, entries in the object header corresponding to ACL entries corresponding to honest users are encrypted under the dummy key pk_0 instead of honest users' keys. The *key-privacy* of the encryption scheme used for object header entries guarantees that $\mathcal{A}$ cannot distinguish $\mathcal{H}_2$ from $\mathcal{H}_3$.

Hybrid $\mathcal{H}_4$. This is the same as $\mathcal{H}_3$, except for the following differences: (1) $\mathcal{F}$ computes ACL' and c and gives those to $\mathcal{S}$ instead of ACL, and (2) $\mathcal{S}$, when creating the corresponding object header to use with the real-world protocol, pads the key list to size c using encryptions of zero under the dummy key pk_0. As before, $\mathcal{S}$ uses its header table to properly interact with the server on behalf of honest users.

The *semantic security* of the encryption scheme used for ACLs guarantees that $\mathcal{A}$ cannot distinguish $\mathcal{H}_3$ from $\mathcal{H}_4$.

Hybrid $\mathcal{H}_5$. This is the same as $\mathcal{H}_4$, except for the following differences: (1) $\mathcal{F}$ maintains its content table and replaces object $\mathsf{contents}$ for untainted objects with $\mathsf{contentIDs}$ and gives only $\mathsf{contentID}$ and ℓ to $\mathcal{S}$ for untainted objects and (2) $\mathcal{S}$ maintains its ciphertext table and uses fake ciphertexts when interacting with $\mathcal{A}$ for operations where $\mathcal{F}$ gives it only $\mathsf{contentID}$ and ℓ.

Although $\mathcal{S}$ now relies on lookups in the ciphertext table based on ciphertexts provided by $\mathcal{A}$ to interact with untainted objects, the lookups are guaranteed to succeed because the object contents

are signed with WSK, which $\mathcal{A}$ does not have for untainted objects. Thus, if the adversary attempts to produce a novel object content ciphertext for which the lookup would fail, for an untainted object, it would have to forge a signature, which it cannot do except with negligible probability due to the *existential unforgeability* of the signature scheme. Recall that, if $\mathcal{A}$ were to produce an object content ciphertext that is not properly signed, $\mathcal{S}$ would abort the operation, as would a real-world client, and return $\perp$ to $\mathcal{F}$.

Once again, *semantic security* of the encryption scheme used to encrypt object contents guarantees that $\mathcal{A}$ cannot distinguish $\mathcal{H}_4$ from $\mathcal{H}_5$.

Hybrid $\mathcal{H}_6$. This is the same as $\mathcal{H}_5$, except for the following differences: (1) $\mathcal{S}$ maintains its anonym table and free token pool and (2) $\mathcal{F}$ maintains its token table and only reveals the anonym to $\mathcal{S}$ when tokens are spent in API calls.

The *blindness* property of the blind signature scheme guarantees that $\mathcal{A}$ cannot distinguish $\mathcal{H}_5$ from $\mathcal{H}_6$. □

A.2 Ghostor's Integrity Guarantee

In this appendix, we state the integrity guarantee provided by Ghostor.

A.2.1 Linearizability

Before we formalize Ghostor's VerLinear guarantee, we define linearizability as a consistency property. Linearizability is well-studied in the systems literature [218, 188], and providing a comprehensive survey of this literature and a fully general definition is out of scope for this dissertation. Here, we aim to define linearizability in the context of Ghostor, to help frame our contributions.

Definition 4 (Linearizability)**.** *Let F be a set of objects stored on a Ghostor server, and let U be a set of users who issue read and write operations on those objects. The server's execution of those operations is* linearizable *if there exists a linear ordering L of those operations on F, such that the following two conditions hold.*

1. *The result of each operation must be the same as if all operations were executed one after the other according to the linear ordering L.*
2. *For every two operations A and B where B was dispatched after A returned, it must hold that B comes after A in the linear ordering L.*

In Ghostor, **an object's digest chain implies a linear ordering** L of `GET` and `PUT` operations, as follows.

Linear ordering L implied by a digest chain. The linear ordering L to which the server commits is based on the digest chain as follows. First, we assign a sequence number to write operations according to the order of their PREPARE digests in the digest chain. Next, we bind each operation to a digest in the digest chain as follows:

- Each read is bound to the digest representing that read.

- A write with sequence number i is bound to the first COMMIT digest corresponding to a write whose sequence number is at least i. **This is either the COMMIT digest for this write, or the COMMIT digest for a concurrent write that wins over this one based on the conflict resolution policy in Section 6.5.4.**

Assuming the digest chain is well-formed, each write will be bound to a COMMIT digest that is after its PREPARE digest and before or at its COMMIT digest. Finally, we generate the linear ordering as follows:

- If two operations are bound to different digests, then they appear in L in the same order as the digests appear in the digest chain.
- If two writes are bound to the same digest, then they are ordered in L according to their sequence numbers.

For example, suppose the digest chain contains

$$(R_1, P_1, R_2, P_2, R_3, C_2, R_4, P_3, R_5, C_1, R_6, C_3, R_7, P_4, R_8, C_4, R_9)$$

where R denotes a read digest, P denotes a PREPARE digest, and C denotes a COMMIT digest. The corresponding linear ordering of operations is

$$L = (R_1, R_2, R_3, W_1, W_2, R_4, R_5, R_6, W_3, R_7, R_8, W_4, R_9)$$

where R denotes a read operation and W denotes a write operation.

A.2.2 Verifiable Linearizability

We begin by stating and proving Theorem 5 below, which specifies the achieved guarantees when some users perform the verification procedure for an epoch. Then, we present the VerLinear property of Ghostor as Corollary 1, a special case of Theorem 5. We use this approach because Theorem 5, despite being a more general statement, has fewer edge cases than Corollary 1, and we feel its proof is easier to understand in isolation. The statement of Corollary 1 maps directly to our informal definition of verifiable linearizability in Section 6.3; the key differences are only that Corollary 1 is explicit that security depends on collision resistance of Ghostor's hash function and existential unforgeability of Ghostor's signature scheme, introduces variables that are useful in the proof, and states the security guarantee as the contrapositive of Guarantee 1.

Theorem 5 (Epoch Verification Theorem)**.** *Suppose that the hash function H used by Ghostor is a collision-resistant hash function [191] with security parameter κ and all users see the same list of checkpoints published to the blockchain. Let $\mathcal{B}$ be a non-uniform adversary that is probabilistic polynomial-time in κ performing an active attack on the server. Let E be a list of consecutive epochs. For each epoch $e \in E$, let U_e be a set of users for whom the verification procedure for a particular object F detected no problems during epoch e, and let O_e be the set of operations performed by those users on F. **If** $U_e \neq \varnothing$ (i.e., U_e is nonempty) for all $e \in E$, **then** there exists, with probability at least $1 - \mu(\kappa)$, where μ denotes a negligible function, a linear ordering L of operations in $O = \bigcup_{e \in E} O_e$ and possibly some other operations reflected in the digest chain, such that for the users in U and their operations O, the following two statements hold.*

1. *The result of each successful operation is the same as if all operations were executed one after the other according to L.*
2. *For every two operations A and B where B was dispatched after A returned, B comes after A in L.*

Proof. We will perform a reduction to show that if there exists an adversary $\mathcal{B}$ that can cause one of the two conditions to be violated, then there exists an adversary $\mathcal{A}$ that can violate the collision-resistance of H with non-negligible probability. For concreteness, suppose that $\mathcal{B}$ performs such an attack with non-negligible probability $\delta(\kappa)$ (so that the condition in the theorem holds with probability $1-\delta(\kappa)$). We will explain how $\mathcal{A}$ can succeed in finding a hash collision with non-negligible probability.

By the nature of the attack, $\mathcal{B}$ is able to violate the property in the theorem statement, while remaining undetected by users in U. Observe that $\mathcal{B}$'s attack must fall into at one of four cases.

1. There exists at least one object such that $\mathcal{B}$ does not commit to a valid digest chain for an epoch, for some honest user.
2. There exists at least one object such that $\mathcal{B}$ commits to a different digest chain for different honest users.
3. There exists an operation on an object $f \in F$ whose result is different from the result that would be obtained by applying the operations one after the other in the linear ordering implied by f's digest chain.
4. There exist operations a and b on the same object, where a was issued after b completed, but a precedes b in the linear ordering implied by the digest chain.

In particular, if $\mathcal{B}$'s attack does not fall into one of these cases, then the locality property proved in Section 3 of [218] guarantees that $\mathcal{B}$'s behavior is consistent with the theorem statement (linearizability of operations in L). We will show that no matter which of the above four cases describes $\mathcal{B}$'s attack, $\mathcal{A}$ can find a hash collision.

Case 1. In this case, $\mathcal{B}$ returns an invalid/malformed linear ordering to a user when the user performs an `obtain_digests` operation. The ordering could be invalid because the digest's signature is missing or malformed, or the digests do not form a well-formed chain. This also includes the case where a user's operation is missing from the digest chain. Because we require that $U_e \neq \varnothing$ for all $e \in E$, this will be detected with probability 1. Therefore, we do not consider this case.[1]

Case 2. In this case, the adversary returns different histories to different users. Because the histories differ, they cannot be the same in all epochs; we consider an epoch e in which they differ. This allows us to confine our argument to a single epoch. In particular, there exist two `obtain_digests` operations on the same object during epoch e, for which $\mathcal{B}$ returns different histories in a way that is not detectable.[2] We define two subcases.

[1] For the purpose of this proof, it does not matter which party signs the digest, only that the signature is not missing or malformed. In the actual Ghostor system, only an authorized user can produce the signature due to the existential unforgeability of the signature scheme; this theorem does not rely on this fact, but Corollary 1 does.

[2] If for all $e \in E$ where the histories differ, only a single call is made to `obtain_digests`, then the server cannot commit to multiple histories, and therefore cannot attack the protocol in this way; therefore, we do not consider this case.

In the first subcase, the leaf of the Merkle tree, containing the hash of the final digest for the object in the epoch, is different for each call. However, given our consistency assumption for the blockchain, each user will see the same Merkle root. Furthermore, because the leaves of the Merkle tree are sorted and each intermediate node indicates the range of objects in each of its children, each node in the root-to-leaf path unambiguously specifies the hash of the next node in the path. Because the first element (root) is the same for the paths returned in each call to `obtain_digests`, but the last element is different, there must be a hash collision somewhere along the path. $\mathcal{A}$ finds this collision.

In the second subcase, both calls to `obtain_digests` see the same Merkle leaf and therefore the same hash of the final digest, but see different digest chains regardless. Observe that the last digest and first digest, for this epoch's digest chain, are fixed based on the checkpoint for this epoch and the checkpoint for the previous epoch, which the client can obtain from the server (to make the argument simpler, we consider the final digest of the previous epoch to also be the first digest of the current epoch). Furthermore, the user knows the hashes of these digests, from the checkpoints on the blockchain. Therefore, if first or last digests of the digest chains returned to both calls to `obtain_digests` differ, then $\mathcal{A}$ can use them to find a hash collision (since their hashes must match the Merkle leaves). If these digests match, then the intermediate digests must differ. To find a collision in this case, $\mathcal{A}$ walks backwards along the digest chains, until they differ. $\mathcal{A}$ can use the digests on each chain, at the point that they differ, to obtain a hash collision.

Case 3. Observe that the result of any committed write is "Success." Therefore, we can restrict this case to *reads that return the wrong value*.

Suppose that a read operation in O_e (for some $e \in E$) returned a value that is not consistent with the linear ordering for the object. In order for the operation to be considered successful, the $\mathsf{Hash}_{\mathsf{data}}$ value in the signed digest received by the client must match the hash of the returned object contents. Furthermore, the verification procedure guarantees that the $\mathsf{Hash}_{\mathsf{data}}$ value in each digest corresponding to a read matches the $\mathsf{Hash}_{\mathsf{data}}$ value in the latest write at that time—it does this by checking that $\mathsf{Hash}_{\mathsf{data}}$ never changes as the result of a read, and that it only changes in the COMMIT digests of winning writes. It follows that the incorrect value returned by the read operation, and the correct value that should have been returned (which was written by the latest write), have the same hash. $\mathcal{A}$ can present these two values as a hash collision.

Case 4. If an operation is missing from the digest chain entirely, this will be detected by the client that issued the operation. We now consider the case where the digests appear in the wrong order. Concretely, let op_1 and op_2 be two operations, where op_2 is issued after op_1 completed. If op_1 is a `PUT`, then d_1 is its COMMIT digest; otherwise, if op_1 is a `GET`, d_1 is the single digest for that `GET`. If op_2 is a `PUT`, then d_2 is its PREPARE digest; otherwise, if op_2 is a `GET`, d_2 is the single digest for that `GET`. Because op_2 is issued after op_1 completed, their digests should unambiguously appear in order in the digest chain: d_1 appears before d_2. Now, suppose d_1 appears sometime after d_2, so that the linear ordering is inconsistent with execution order. In this case, $\mathcal{A}$ waits until the users have run the verification procedure, and then rewinds $\mathcal{B}$'s state to a point after $\mathcal{B}$ has committed op_1, but before op_2 has been issued. The client places a fresh nonce in d_2 this time around, but otherwise execution is resumed as before. $\mathcal{A}$ waits until the user runs the verification procedure again, and it compares the digest chains produced by $\mathcal{B}$'s execution both times. Because all that

changed is the client's nonce in d_2, and it is taken from the same uniform random distribution, $\mathcal{B}$'s probability of performing a successful attack is still non-negligible. So the probability that $\mathcal{B}$ performed a successful attack in both distributions is non-negligible ($\delta(\kappa)^2$). In this case, $\mathcal{A}$ walks the digest chains backward starting at d_1; the digest chains must differ at some point, because d_2 precedes d_1 in the first history, d_2 has a different random nonce in the second history, and the digest for d_1 is the same in both histories. This way, $\mathcal{A}$ can obtain a hash collision with non-negligible probability. □

Although the two conditions in Theorem 5 are the same as those in Definition 4, Theorem 5 does *not* guarantee linearizability of operations in O (operations performed by users in U). This is because the linear ordering L in Theorem 5 includes *additional* operations in the system beyond those in O, which could be digests that the server replayed or operations performed by users who did not run the verification procedure. This motivates us to state Corollary 1, which specifies under what conditions a set of users can be sure that their operations were processed in a linearizable way. Because our definition is now in line with linearizability (Definition 4), we can leverage the locality property of linearizability [218] to state the corollary in terms of a single object.

Corollary 1 (Verifiable Linearizability)**.** *Suppose that the hash function H used by Ghostor is a collision-resistant hash function [191], all users see the same list of checkpoints published to the blockchain, and the signature scheme is existentially unforgeable [191]. For any adversary probabilistic polynomial-time in κ, any object F, and any list E of consecutive epochs: suppose that for each epoch $e \in E$, the set U_e of users who ran the verification procedure on F during epoch e (1) is nonempty (i.e., $U_e \neq \varnothing$) and (2) contains all users who wrote the object F during epoch e (and possibly other users too). With probability at least $1 - \mu(\kappa)$, where μ denotes a negligible function,* ***if*** *no user detects a problem when running the verification procedure,* ***then*** *the server's execution of operations in $O = \bigcup_{e \in E} O_e$ is linearizable, where O_e is the set of operations performed by users in U_e during epoch e.*

Proof. By Theorem 5, we know that there exists a linear ordering L containing all operations in O plus some other operations on F (that are reflected in the digest chain) such that Properties #1 and #2 in the statement of Theorem 5 hold for operations in O, with respect to L. Because each U_e contains all users who wrote f during epoch e, and the signature scheme is existentially unforgeable, we know that all operations in L that are not in O must be reads. Let ℓ denote the subset of L consisting only of operations in O. Observe that Properties #1 and #2 in the statement of Theorem 5 also hold for the operations in O *with respect to* ℓ. This is because (1) the only operations in L but not ℓ are reads, so the result of each operation in ℓ, when operations are executed one after the other, is the same for both L and ℓ, and (2) ℓ preserves the relative ordering of operations in L (i.e., any two operations that appear in ℓ appear in L *in the same order*.). Because ℓ contains only operations in O and it satisfies Properties #1 and #2, it fulfills Definition 4. Therefore, the execution of operations in O is linearizable. □