SECURITY AND PRIVACY FOR THE MODERN WORLD

by
Tushar M. Jois

A dissertation submitted to the Johns Hopkins University in conformity with the requirements for the degree of Doctor of Philosophy.

Baltimore, Maryland
March 2023

© 2023 Tushar M. Jois
All rights reserved.

Abstract

The world is organized around technology that does not respect its users. As a precondition of participation in digital life, users cede control of their data to third-parties with murky motivations, and cannot ensure this control is not mishandled or abused. In this work, we create secure, privacy-respecting computing for the average user by giving them the tools to guarantee their data is shielded from prying eyes. We first uncover the side channels present when outsourcing scientific computation to the cloud, and address them by building a *data-oblivious virtual environment* capable of efficiently handling these workloads. Then, we explore stronger privacy protections for interpersonal communication through *practical steganography*, using it to hide sensitive messages in realistic cover distributions like English text. Finally, we discuss *at-home cryptography*, and leverage it to bind a user's access to their online services and important files to a secure location, such as their smart home. This line of research represents a new model of digital life, one that is both full-featured and protected against the security and privacy threats of the modern world.

Thesis Readers

Dr. Aviel D. Rubin (Primary Advisor)
Professor
Department of Computer Science
Johns Hopkins University

Dr. Matthew Green
Associate Professor
Department of Computer Science
Johns Hopkins University

Dr. Gabriel Kaptchuk
Research Assistant Professor
Department of Computer Science
Boston University

Dr. Michael Rushanan
Director of Medical Security
Harbor Labs

To Amma and Appa.

Olleya hesaru tandiddene.

Acknowledgements

After five years in a PhD, and eight since I first started at Hopkins, it's hard not to be reflective. A number of people have touched my life in the course of these years, and I want to take this opportunity to attempt to thank as many of them as I can.

The first person I have to thank is my advisor, Avi Rubin. Who knew a fateful email, inquiring about research as a sophomore, would turn into this? Avi has had his finger on the pulse of this field for decades, and has been an invaluable resource as I build my own career. In the years Avi has been my advisor—both undergraduate and graduate—he has been reliable, responsive, and always ready with a anecdote or parable fitting to my current dilemma. He is the model for the type of professor I hope to be some day. Think of me when you're boating the Great Loop?

I would like to next thank Gabe Kaptchuk for introducing me to the trenches of academia, and for shepherding me through them. Gabe was the first person I interacted with in the lab, and he has always treated me with respect and class, even when pestered with incessant questions. His enthusiasm for all things research and teaching is infectious, and played a huge part in me deciding to become an academic. Don't worry, I'm still going to keep bothering you.

I would like to thank Matt Green for effectively adopting me as a member of his lab when I first started my PhD. Working with Matt gave me exposure to and appreciation for the nuances of cryptography, which I have now used in every single one of my projects. I'd like to think I've come a long way from confusing the RSA and discrete logarithm

problems in my first-year Practical Crypto class, but I suppose you'd be the best judge of that.

I also want to thank Mike Rushanan, who has mentored my journey working in industry. Shadowing him in client meetings has taught me how best to bridge the technical concepts of academia to the practical concerns of end users and businesses. Mike's affable demeanor, even in the face of tense situations, is something I aspire to match. I probably won't ever be as nice as you are, though.

I am very grateful that I had the opportunity to learn from the some of the best professors in the world during my PhD. Thank you to Yinzhi Cao, Matt Green, Ryan Huang, Xin Jin, Abhishek Jain, and Avi Rubin for instilling in me the knowledge of areas of work beyond those of my own focus, giving me the tools to extend my research into new directions.

I have had the pleasure of learning alongside wonderful fellow students and researchers in the lab over the years. I want to thank Atheer Almogbil, Gabby Beck, Sofia Belikovetsky, Pedro Branco, Alishah Chator, Arka Rai Choudhuri, Harry Eldridge, Madeline Estey, Aarushi Goel, Aditya Hegde, David Inyangson, Zhengzhong Jin, Gabe Kaptchuk, Stephan Kemper, Darin Khan, Logan Kostick, Claudia Moncaliano, Jonathan Prokos, Dave Russell, Momo Steele, Pratyush Ranjan Tiwari, Gijs Van Laer, Olivia Wu, and Max Zinkus. Our lab culture is unparalleled, and it's because of the vivacity and kindness that each of you brought to Malone (or, during COVID, Zoom) every day. Walking down to the corner of St. Paul and 32nd street for lunch, and the raw, unbridled indecision that follows, is my favorite daily tradition, and one that will be sorely missed. While I may not have had a chance to have a paper with everyone yet, there's always the next project, right? Keep in touch.

My research would not be possible without the generous support of the National Science Foundation through grant number 1955172, Security and Privacy in the Lifecycle of IoT for Consumer Environments. I would like also to thank the students, professors,

and staff of the SPLICE team, with a particular shout-out to the inimitable Tina Pavlovich for being the greatest project coordinator in the history of science, if not humanity.

During the course of my PhD, I was fortunate to be hosted for two summer visits, one with Carl Gunter and Chris Fletcher at the University of Illinois, Urbana-Champaign, and the other with David Kotz and Tim Pierson at Dartmouth College. Carl and Chris at UIUC gave me a crash-course in hardware security and side channels, resulting in my first completed research project, with fellow student Hyun Bin "HB" Lee. At Dartmouth, with the students and researchers of the Kotzgroup, Nurzaman Ahmed, Mounib Khanafer, Tomi Oluwaseun-Apo, Matthew Wallace, and Chixiang Wang, I deepened my knowledge of software, network, and IoT security, and was treated to a spectacular visit to one of the most beautiful places I've been, the Moosilauke Ravine Lodge. I would like to especially thank HB, for joining me as a visitor at Dartmouth and for all of the adventures—both research and road-trip-adjacent—we've had since.

I would like also to thank Harbor Labs, its employees, and its CEO, Nick Yuran, for hosting me for multiple internships. A special thanks to the leads at at Harbor Labs, Paul Martin and Mike Rushanan, for sharing with me their experiences in research-oriented security consulting, and their love of Jumbo Seafood, Pikesville, MD.

Even at this early stage of my career, I have had the opportunity to work with spectacular individuals in this field and beyond on publications. I would like to thank my collaborators Gabby Beck, Sofia Belikovetsky, Yinzhi Cao, Joe Carrigan, Alishah Chator, Neil Fendley, Chris Fletcher, Matt Green, Carl Gunter, Khir Henderson, Gabe Kaptchuk, Mounib Khanafer, Logan Kostick, HB Lee, Claudia Moncaliano, Jonathan Prokos, Avi Rubin, Roei Schuster, Eran Tromer, and Max Zinkus.

I also want to thank all of the students from all of my classes over the years for teaching me how to be a better educator and mentor.

I would not have made it this far with my sanity still (mostly) intact without my

non-academic support system. I owe a debt of gratitude to the lads of Baltimore, with whom I spent many a night arguing who failed the quest, where to build the research station, or how insane jet hammer is. These times never failed to alleviate the stresses of grad school, as paradoxical as it may seem. To my best friend and roommate of seven years Rohan Panaparambil, I just wanted to say, thanks. Partner.

I want to thank my fiancée and future wife Shreya Suresh for supporting me through my deadline-day (and, frankly, everyday) panics. I want to thank my younger sister Tunga Jois for offering me wisdom and counsel well beyond her years, and for tolerating her cringe *anna*.

Finally, and most of all, I want to thank my parents, Muralidhara and Vidya Jois. Words cannot express the level of privilege I have had to be raised by two incredibly caring, loving, and selfless individuals, and the amount of gratitude I have as a result. My parents have worked hard every day to allow their children to pursue their dreams.

And here I am today, living mine.

Contents

Abstract . . . . **ii**

Thesis Readers . . . . **iii**

Dedication . . . . **iv**

Acknowledgements . . . . **v**

Contents . . . . **ix**

List of Tables . . . . **xvi**

List of Figures . . . . **xix**

Chapter 1 Introduction **1**

1.1 Threats Considered 2

1.1.1 Side Channels in Outsourced Computation 2

1.1.2 Censorship of Encrypted Communication 3

1.1.3 Loss of Cryptographic Secrets 4

1.2 Proposed Approach 5

1.2.1 Dissertation Outline 6

Chapter 2 Data-Oblivious Scientific Computation **7**

2.1 Introduction 7

2.1.1 Our Solution: DOVE 8
2.1.2 Contributions 9
2.2 Background 9
2.2.1 Programming in R 9
2.2.1.1 Computation in R 9
2.2.1.2 Not Applicable (NA) 10
2.2.2 Microarchitectural Side-Channel Attacks 10
2.2.3 Enclave Execution and Intel SGX 10
2.2.3.1 Side-Channel Amplification 11
2.2.4 Threat Model 12
2.2.5 Data-Oblivious Programming 12
2.3 The (Lack of) Data-Obliviousness of R 13
2.3.1 Case Study 13
2.3.2 Example Walkthrough 15
2.3.3 Instruction-Level Analysis 16
2.3.3.1 Static Opcode Analysis 17
2.3.3.2 Dynamic Execution Trace Analysis 17
2.3.4 Intel PCM Analysis 18
2.3.5 Discussion 21
2.4 The Data-Oblivious Virtual Environment 22
2.4.1 Design Principles 22
2.4.2 Overview 23
2.4.3 Data-Oblivious Transcript (DOT) 25
2.4.3.1 Data Creation, Types and Operations 28
2.4.3.2 Control Flow 29
2.4.4 Frontend 30
2.4.4.1 Construct-Specific Handling 32

2.4.5 Backend 33
2.5 Experimental Evaluation 35
2.5.1 Correctness and Expressiveness 35
2.5.1.1 Unit Tests 35
2.5.1.2 PageRank 36
2.5.1.3 Evaluation Scripts 36
2.5.2 Data-Obliviousness 37
2.5.2.1 Static Opcode Analysis 38
2.5.2.2 Dynamic Execution Analysis 40
2.5.2.3 Intel PCM 40
2.5.3 Efficiency 42
2.5.3.1 Programs with Quadratic Space Complexity 45
2.5.3.2 Statistical Programs 45
2.5.3.3 Remaining Programs 46
2.5.4 Lessons Learned 46
2.5.4.1 Automating Expressiveness 46
2.5.4.2 Data-Oblivious Statistics 48
2.5.4.3 Efficient Looping 48

Chapter 3 Secure Steganography for Realistic Distributions 50
3.1 Introduction 50
3.1.1 Overcoming the Shortcomings of Existing Techniques 51
3.1.1.1 Generative Models as Steganographic Samplers 52
3.1.1.2 Channels with High Entropy Variability 53
3.1.2 Contributions 54
3.1.2.1 Deployment Scenario 55
3.1.3 Limitations 56
3.2 Background and Related Work 57

3.2.1 Classical Steganography 57
3.2.2 Current Steganography in Practice 60
3.2.3 Generative Neural Networks 61
3.3 Definitions 62
3.3.1 Symmetric Steganography 62
3.3.1.1 Correctness 63
3.3.1.2 Security 63
3.3.2 Ranged Randomness Recoverable Sampling Scheme 64
3.3.2.1 Correctness 65
3.3.2.2 Coverage 65
3.4 Adapting Classical Steganographic Schemes 66
3.4.1 Characterizing Real Distributions 66
3.4.1.1 Adaptation 1: Entropy Bounding 66
3.4.1.2 Adaptation 2: Variable Length Samples 69
3.5 More Efficient Symmetric-Key Steganography 72
3.5.1 Intuition 72
3.5.2 Meteor 74
3.5.2.1 Construction 75
3.5.2.2 Correctness 78
3.5.2.3 Security 78
3.5.2.4 Efficiency 79
3.6 Evaluation 81
3.6.1 Implementation Details 81
3.6.2 Optimizations 84
3.6.2.1 Optimization 1: Reordering the Distribution 84
3.6.2.2 Optimization 2: Compressing with Native Embedding . . 87
3.6.3 Results 87

3.6.3.1 Model Performance 88
3.6.3.2 Encoding Statistics 88
3.6.3.3 Optimizations 89
3.6.3.4 Mobile Benchmarks 90
3.6.4 Sample Model Outputs 91
3.7 Comparison to NLP-Based Steganography 97
3.7.1 Comparative Analysis 99

Chapter 4 Building At-Home Cryptography 101
4.1 Introduction 101
4.1.1 Prior Attempts 102
4.1.1.1 Re-using Devices for At-Home Cryptography 103
4.1.2 SocIoTy 103
4.1.3 Contributions 105
4.2 Background 107
4.2.1 Smart Homes 107
4.2.2 Pseudorandom Functions (PRFs) 107
4.2.2.1 (Threshold) Distributed PRFs 108
4.2.3 Two-Factor Authentication (2FA) 109
4.2.3.1 Compelled Access 109
4.3 Designing At-Home Cryptography 110
4.3.1 Case Studies 111
4.3.1.1 Use Case 1: The Remote Worker 111
4.3.1.2 Use Case 2: The Outpatient 112
4.3.1.3 Use Case 3: The Foreign Correspondent 112
4.3.2 Design Goals 113
4.3.2.1 Functionality 113
4.3.2.2 Deployability 113

4.3.2.3 Security . . . 114
4.3.3 Threat Model . . . 115
4.3.3.1 Compelled Access Adversaries . . . 116
4.3.3.2 Local Network Adversaries . . . 116
4.4 SocIoTy . . . 117
4.4.1 Preliminaries . . . 117
4.4.1.1 Choosing the Correct Cryptographic Primitive . . . 118
4.4.1.2 Layering Security . . . 118
4.4.2 Protocol Description . . . 119
4.4.2.1 Setup . . . 121
4.4.2.2 Authentication . . . 121
4.4.2.3 Encryption . . . 122
4.4.3 Security Analysis . . . 123
4.4.3.1 Extensions . . . 123
4.4.4 Deployment Flexibility . . . 124
4.4.4.1 Devices to Use . . . 124
4.4.4.2 Multi-User Smart Homes . . . 125
4.4.4.3 Network Structure . . . 125
4.4.4.4 Server Interface . . . 126
4.4.5 Instantiating the TDPRF . . . 126
4.5 Evaluation . . . 127
4.5.1 Implementation . . . 127
4.5.2 Microbenchmarks . . . 128
4.5.2.1 PartialEval . . . 129
4.5.2.2 Gen and Recon . . . 129
4.5.3 Scalability Benchmarks . . . 132
4.5.4 End-to-End Deployment . . . 136

4.6 Related Work 140
4.6.1 Location-Based Cryptography 140
4.6.2 Hardware Security Modules (HSMs) 142
4.6.3 Security via IoT Devices 142

Chapter 5 Conclusion **144**

References **146**

List of Tables

2.1 The associated x86-64 instruction counts for different permutations of `x1` and `x2` fed as input to **&** in R. 19

2.2 Intel PCM Functions used for dynamic analysis. 19

2.3 DOVE functions/operations. Functions in group "DOT/Core" are implemented directly in the DOVE backend and are included in the DOT semantics. Functions in the group "Supplemental" are implemented using operations in "DOT/Core" and exposed to the user as library functions. Safe functions require a data-oblivious implementation in the backend as they may receive pseudonyms as operands. Unsafe functions do not require a data-oblivious implementation, but can only take non-pseudonyms (non-sensitive) data as operands. 31

2.4 All x86-64 opcodes that operate on sensitive data in the leaf functions of DOVE. Those marked with * are those not found in libFTFP. 39

2.5 Absolute runtimes and sizes of the evaluation programs. Programs marked with an * were run on a reduced dataset due to test system limitations. Program `iES` calls `EHHS`, so we include the lines of code from `EHHS` when measuring lines of code for `iES`. FE are measurements for frontend, NEBE are for measurements with backend without SGX, and EBE are for the backend with SGX. F indicates the use of libFTFP, the data-oblivious floating point arithmetic library that we used on our DOVE implementation. LoC stands for Lines of Code for the original R program whereas DOT size represents the size of the counterpart DOT file in bytes. Finally, the DOT overhead represents the relative overhead of the DOT's file size relative to the size of the original R program. 43

3.1 Performance results for model load encoding using the method of [98] and resampling, averaged over 30 runs. The message being encoded is the first 16 bytes of Lorem Ipsum. 72

3.2 Model statistics for encoding a 160-byte plaintext. Timing results reflect model load, encoding, and decoding combined. 90

3.3 Performance measurements for Meteor on the GPT-2 by device for a shorter context. Times are provided in seconds. 90

3.4 Comparative distribution statistics for samples from neural steganography algorithms in prior NLP work, with random sampling as a baseline. "N/A" indicates that a metric is not relevant for an algorithm. 99

4.1 Hardware specifications of test devices as well as examples of comparable IoT devices. 128

4.2 Average runtimes for an evaluation of **PartialEval**, both without and with authenticated encryption (**AE**). All times are in milliseconds. 129

4.3 Microbenchmarks for Gen and Recon on the ESP32 devices over varying configurations of total number of parties n and reconstruction threshold t. All times are in milliseconds. Although we do not expect users to use the ESP32 for Gen or Recon, our implementation is nonetheless efficient enough for this purpose. 133
4.4 Protocol execution time CoAP over all evaluated configurations of total number of parties n and reconstruction threshold t. All times are in milliseconds. The large spike in μ, σ at $n \geq 10, n = t$ is likely due to faulty hardware. 135
4.5 A comparison of related work with similar goals to those of SocIoTy. 141

List of Figures

2.1 R code snippet. `geno` is a sensitive diploid dataset. 16

2.2 The R interpreter implementation of the `&` operator. 18

2.3 Number of cycles taken to run one million iterations of 0 & 0 and 1 & 0. Each boxplot represents 100 measurements of each expression. 20

2.4 High-level overview of DOVE. Bold-face arrows between nodes represent communication over (mutually-authenticated) TLS, while thinner ones are intra-process communication within a component. Shading indicates the location of our trusted computing base (TCB). 25

2.5 A DOT (left) and its associated R program (right). The matrix `x` corresponds to the pseudonym `$1` in the DOT, and the loop index `i` with `\1`. ① corresponds to line 1 of the program, ② the `for` loop on line 3, ③ the `if` statement on line 4, and ⑥ the assignment in line 5. Intermediate values are stored in variables marked with `%`, and constants are declared using `#`. 26

2.6 The grammar for the DOT language, presented in extended Backus–Naur form. **`digit`** consists of 0-9, **`nonzero-digit`** of 1-9, and **`alpha`** of A-Z and of a-z. 27

2.7 The DOVE-compatible implementation of PageRank in R. 36

2.8 Intel-syntax assembly for `BitwiseAndOp::call`, the DOVE backend equivalent of `&` in R. This snippet has been lightly edited for clarity. 38

2.9 Cycle count measurements for runtime Intel PCM analysis against Line 1 of Figure 2.1. Plots above are measurements from vanilla R and plots below are from DOVE. The plots on the left are tested against varying proportions of `NA`, and plots on the right are tested against varying proportions of `0`. . 41

2.10 Performance evaluation results for the evaluation programs. Each stacked bar represents a measurement for each program. Each stack represents relative overhead of DOVE against vanilla R caused by generic data-oblivious computation, libFTFP and SGX from left to right. Programs marked with * run on reduced dataset due to machine limitations. 47

3.1 The public-key steganography scheme from [98]. PseudorandomPKEncrypt is the encryption routine for a pseudorandom, public-key encryption scheme. Sample randomly selects an token from the covertext space according to the distribution $\mathcal{D}$. 59

3.2 Entropy of GPT-2 output distributions. Each datapoint computed as Shannon entropy of the output distribution after sampling a certain number of tokens. Then, a random token is sampled from that distribution and appended to the context. Different colors represent different runs starting with the same context and different randomness. 67

3.3 Binned probability of selecting the tokens included in the final stegotext using entropy bounding with a value of 4.5 and the GPT-2 model. The stegotext tokens clearly come from a different distribution. Note that baseline tokens were only sampled from events above the entropy bound. . 68

3.4 Binned probability of selecting the tokens included in the final stegotext variable length sampling. Although there is slight variation in the distributions, there is no clear difference between the stegotext and the baseline. Moreover, this method is proved secure in [98]. 70

3.5 An overview of the encoding strategy for Meteor. In each iteration of Meteor, a new token (shown in green) is selected from the probability distribution created by the generative model. Depending on the token selected, a few bits (shown in red) can be recovered by the receiver. The stegotext above is real output from the GPT-2 model. 75
3.6 Subroutine algorithms for Meteor 76
3.7 Symmetric steganography algorithms for Meteor 77
3.8 Bits of throughput by starting location for an interval i with size $p_i = \frac{1}{4} - \epsilon$, for some small ϵ. The expected throughput can be computed as the average of this function, *i.e.* $\mathsf{Exp}(p_i) \geq (2)(\frac{1}{4} - \epsilon)(0) + (2)(\frac{1}{4} - \epsilon)(1) + (2^2)(\epsilon)(2) = \frac{1}{2} - 6\epsilon$ 80
3.9 RRRSS algorithms for the GPT-2 model. $\mathcal{T}$ is an array of possible next tokens and $\mathcal{P}$ is the probability associated with each of these tokens. 82
3.10 An illustrative example of the impact of reorganizing a distribution. r_0 has 3% of the total probability density, while r_1 and r_2 have 48% and 49% respectively. Because $2^{-6} < .03 < 2^{-5}$, r_0 can encode 5 bits of information when located at the beginning or end of the distribution. In orderings (a) and (c), one of the larger intervals crosses the 50% line, meaning $\mathsf{LenPrefix}(\cdot) = 0$. When the smallest interval is placed in the middle, the total expected throughput of the distribution rises. 85

3.11 An overview of our reorganization algorithm. This distribution has entropy 1.16, so we create $2^2 = 4$ buckets. In (1), we place the largest interval r_1 into bucket 0, overflowing its value through most of bucket 1. Note that r_1 could have been placed in bucket 2; in general, we break ties by taking the earlier bucket. In (2), r_2 can be placed either in bucket 1, overflowing into the following buckets, or placed in bucket 2, overflowing into bucket 3. To maximize $\mathsf{LenPrefix}(r_2)$, we place it in bucket 2. Finally, in (3), we note that r_0 will not fit in bucket 3, so it must be placed in bucket 1. The pushes the later intervals, in this case r_2 down to make sufficient space. 86
3.12 Comparison of plaintext length versus time to run encoding and decoding for different Meteor models. $R = 0.9745$ (GPT-2), 0.9709 (Wikipedia), 0.9502 (HTTP Headers) 89
3.13 The 160-byte plaintext used for the model outputs in this section. 91
3.14 Snippet of Meteor encoding of the first 160 bytes of Lorem Ipsum as generated by the Wikipedia model. The output was truncated to fit. 92
3.15 The "Dinosaur" context and associated Meteor encoding of the first 160 bytes of Lorem Ipsum as generated by GPT-2. This is the context used in Figure 3.5. 93
3.16 The "Washington" context and associated Meteor encoding of the first 160 bytes of Lorem Ipsum as generated by GPT-2. This is the encoding used throughout the benchmarks in Section 3.6. 94
3.17 Snippet of Meteor encoding of Figure 3.13 as generated by the HTTP Headers model. The output was truncated to fit. 95
3.18 iPhone X screenshot of Meteor encoding of the first 160 bytes of Lorem Ipsum as generated by the GPT-2 model. Generated text is highlighted, and context is unhighlighted. 96

4.1 An overview of our SocIoTy, which uses a PRF built from IoT devices to provide at-home cryptographic services. 106
4.2 An overview of our threat model for SocIoTy. 115
4.3 The Setup workflow of SocIoTy 119
4.4 The Authentication workflow of SocIoTy. 120
4.5 The Encryption workflow of SocIoTy. 120
4.6 Microbenchmarks for Gen on different test devices over varying configurations of total number of parties n and reconstruction threshold t. 130
4.7 Microbenchmarks for Recon on different test devices over varying configurations of total number of parties n and reconstruction threshold t. 131
4.8 Protocol execution time over CoAP for varying configurations of total number of parties n and reconstruction threshold t. 134
4.9 End-to-end OTP generation time using our iOS app for varying configurations of total number of parties n and reconstruction threshold t. 137
4.10 A Simulator screenshot of our iOS app. Note that all benchmarks were performed with a hardware iPhone X. 138

Chapter 1

Introduction

Broadly defined, computer security is the study of intrusion detection and prevention on software and systems. Privacy is a related goal, allowing information to remain hidden in the presence of prying eyes. These two properties are less clear fields of study, and more ideals; in the hunt for perfection, researchers are developing innovations – from the stack canary to symmetric ratcheting – that are practical, efficient, and secure primitives with wide-ranging impact on everyday people and systems.

During this period of progress, the democratization of computing has led to an explosive growth in productivity for society. Gone are the days of ARPANET, teletypes, and any gatekeepers to the wonders of computing. A user today could be anyone, and have any level of experience. Users are leveraging computing in myriad new ways, but security primitives do not always conform to these use cases. Many primitives were largely designed for broad threat models and simple adversary types, and for not specialized applications.

Unfortunately, the modern world is organized around complex technology that does have its users' best interests in mind. Users cede control of their data and computing to third-parties to utilize the online services. Incentive structures typically prioritize these service vendors instead of users, and these imbalances in trust and power result in breaks in security and privacy.

1.1 Threats Considered

The security and privacy issues that arise in everyday computing are varied in their scope and impact. We describe them in the context of three areas: outsourced computation, secure communication, and cryptographic secrets.

1.1.1 Side Channels in Outsourced Computation

Recent commercially-available Trusted Execution Environments (TEEs) such as Intel SGX [56, 108] and ARM TrustZone [8] have enabled significant progress towards the outsourcing of secure computation. Consider for example three competing drug companies investigating genomic factors for bipolar disorder. These companies would like to share their proprietary genome data and run a controlled study that releases only agreed-upon information to the three participants. TEEs enable such use cases, without requiring trust in remote administrator software stacks such as operating systems, using a combination of hardware-level isolation and cryptography.

The long-term vision pursued by TEE-based software systems (e.g., [19, 46]) is to bring TEE-level security to the masses where it can be used by data scientists familiar with existing high-level languages such as R, Ruby, and Python, but who may not have much background in security [38].

Here, we face a challenging problem. To achieve complete security from untrusted software, it is well known that TEE software must be hardened to block a plethora of microarchitectural side channels (e.g., [30, 196, 217, 231]). Yet, existing software-based techniques to block these channels—coming from a rich line of research in data-oblivious/constant-time programming [22, 53, 145, 170]—fall short of protecting existing high-level language stacks such as R, Ruby and Python. Specifically, these techniques typically require experts to manually code core routines [22, 23], require the use of custom domain-specific languages [37, 190], or only apply to close-to-metal compiled

languages [145, 170].

Modern high-level languages, however, require complex stacks to support interpreted execution, just-in-time compilation, etc. As a case-in-point, the popular R stack features almost a million lines of code written in a combination of C, Fortran, and R itself [167]. Subtle issues in any of this code create security holes.

1.1.2 Censorship of Encrypted Communication

The past several years have seen a proliferation of encrypted communication systems designed to withstand even sophisticated, nation-state attackers [162, 223]. While these systems maintain the confidentiality of plaintext messages, the data transmitted by these tools is easily identifiable as encrypted communication. This makes these protocols easy targets for repressive regimes that are interested in limiting free communication [52, 79]: for example, using network censorship techniques such as those practiced by countries like China [80, 169, 189]. Concrete attempts to suppress the encrypted communication technologies used to evade censors are now underway. For example, China's Great Firewall (GFW) not only prevents users from accessing content deemed subversive, but it also actively detects and blocks encryption-based censorship circumvention technologies such as Tor [63, 171, 208].

In regimes where cleartext communication is expected, the mere *use* of encryption may be viewed as an indication of malicious or subversive intent. To work around blocking and avoid suspicion, users must make their communications look mundane. For instance, Tor users in China have begun to leverage steganographic techniques such as ScrambleSuit/obfs4 [225], SkypeMorph [142], StegoTorus [221], TapDance [228, 229], and Format-Transforming Encryption [67]. These techniques embed messages into traffic that censors consider acceptable.

While the current generation of steganographic tools is sufficient to evade current censorship techniques, these tools are unlikely to remain a sustainable solution in the

future. Some tools do provide strong cryptographic guarantees [101, 142, 216], but this is achievable only because they encode messages into (pseudo-)random covertext channels, *i.e.*, replacing a random or encrypted stream with a chosen pseudorandom ciphertext. Unfortunately, there is no guarantee that such channels will continue to be available: a censor can systematically undermine such tools by preventing the delivery of encrypted traffic for which it does not have a suitable trapdoor, (*i.e.*, an access mechanism), or by selectively degrading the quality of encrypted channels. An audacious, repressive regime could even consider *all encryption* to be subversive, and drop all packets not explicitly recognizable as meaningful plaintext. Rigorous studies of the capabilities of the current GFW focus on other techniques [70, 71, 134, 209], but there is anecdotal evidence that encryption suppression has begun to occur [24], including the blocking of some TLS 1.3 connections [29].

1.1.3 Loss of Cryptographic Secrets

Mobile devices have quickly become users' most important trusted computing base. Users rely on mobile devices to authenticate and interact with services that perform sensitive tasks, *e.g.*, online banking, file storage, and telehealth. These tasks are often secured using a combination of passwords and locally-stored cryptographic secrets, *e.g.*, one-time passwords (OTPs) generated by a smartphone app for two-factor authentication (2FA).

The convenience afforded by mobile computing is accompanied by a commensurate increase in risk. Mobile devices are highly portable, allowing them to be easily lost or stolen. Once a mobile device is taken, any cryptographic material on the device could be extracted [251]. This would allow an adversary to impersonate the user and access their services—a catastrophic breakdown in online security and privacy. A corporate spy, for instance, could use extracted 2FA OTPs to connect to a rival company's internal VPN. This threat is particularly dire when the user needs to keep their data private from law enforcement agencies with access to software that can be used to circumvent on-device

security measures. For example, border police could decrypt files from a user's cloud storage, inspecting it for content deemed subversive.

1.2 Proposed Approach

In this dissertation, we explore technologies that allow users to take back control of their everyday computing. Each of the above threats concerns an individual aspect of a user's security and privacy. This work provides a solution to each threat, tailored to the considerations of the problem and efficient enough to be deployed in practice.

A user at work should not have to think about if their sensitive cloud computing tasks are not leaking information. So, we propose a trusted hardware-based architecture that ensures code is free of such leaks before execution. A user should not have to worry that their communications with others will be blocked due to censorship. We therefore create a system that hides messages in seemingly-innocuous cover distributions. A user should not have to be concerned with losing their authentication or encryption keys along with their smartphone. Thus, we design a way for users to stash their secrets at home, re-using devices they already own.

Overall, our proposal represents a step towards assuring users their data is always working on their behalf. We consider each solution individually, as users have different expectations of functionality, security, and efficiency based on the context of the application. But, in each case, users are afforded guarantees based in proven systems and cryptographic models, while maintaining the level of performance and clarity of interface that they expect. Thus, all of the solutions fit together in aggregate to give users a more trustworthy digital life.

1.2.1 Dissertation Outline

We now give a brief overview of the components of this dissertation. First, in Chapter 2, we analyze the side channels present in scientific computation, and provide a solution that transforms vulnerable code into side-channel-free data-oblivious code. Next, in Chapter 3, we build a practical steganographic system, capable of embedding secret information into realistic generative model distributions, like English text. Then, in Chapter 4, we show how Internet-of-Things devices can be used in concert to provide cryptographic services, tying a user's secrets to their smart home. Finally, in Chapter 5, we provide some concluding thoughts on the solution model we propose.

Chapter 2

Data-Oblivious Scientific Computation

The goal of this chapter is to extend data-oblivious/constant-time techniques to apply to existing high-level, interpreted languages, thus enabling TEE-level security for non-experts when outsourcing scientific computation.

2.1 Introduction

The key strategy and insight is this: *if key observable features of a computation are truly independent of sensitive data, then that computation can be carried out with a collection of stand-ins ("pseudonyms") for the data*.

To capitalize on this idea, we perform computation in two phases. In the first phase, we run the target computation on pseudonyms in the chosen high-level language, like R or Python. Since there is no sensitive data present, this stage cannot leak sensitive information. We instrument the programming stack so that this evaluation on pseudonyms outputs what we call a "Data-Oblivious Transcript (DOT)". The DOT is akin to a straight-line code representation of the original program, i.e., the transcript of operations performed when the program is evaluated on the pseudonyms. In the second phase of our computation, we evaluate the DOT on a small Trusted Computing Base (TCB) that runs within a TEE. This TEE contains the sensitive data, which is used in place of the pseudonyms. Protecting sensitive data *after* the DOT is constructed is relatively

straightforward. Since the DOT is similar to straight-line code, the TEE need only apply simple transformations to evaluate it in a data-oblivious fashion on real hardware. In the worst case, where the original computation was actually data dependent on the pseudonyms, the resulting computation in the TEE may be functionally incorrect but leaks no sensitive information.

Conceptually, the DOT plays a role similar to a compiler intermediate representation. A key benefit of this decoupled approach is that only the backend (importantly, *not* the frontend) is part of the TCB. This provides a powerful strategy for protecting complex, high-level programming stacks against side channel attacks. In addition to a reduced TCB, the decoupling provides modularity and extensibility benefits similar to those found in modern compilers. For example, to add support for a new high-level language, we need only change frontend code. Likewise if a security vulnerability is found in the TEE, or we wish to deploy different TEEs to protect execution for different processor microarchitectures, we need only change backend code.

2.1.1 Our Solution: DOVE

Putting it all together, we design and implement an instance of the above architecture, called the "Data-Oblivious Virtual Environment (DOVE)". Our proof-of-concept DOVE implementation[1] is two-fold: a DOVE *frontend* that translates programs written in the R language to a DOT representation, and a DOVE *backend* that evaluates the DOT on sensitive data inside of an Intel SGX enclave. We validate DOVE in four domains: correctness, expressiveness, data-obliviousness, and efficiency. We experimentally compare DOVE's results in these domains to those of base R, using a third-party library of genomics analysis algorithms written in R [39] applied on a real-world genomic dataset consisting of three populations of honeybees [13].

[1] Code and benchmarks are available at `https://github.com/dove-project/`

2.1.2 Contributions

1. We identify a number of subtle side-channel vulnerabilities in the R language.
2. We design DOVE, the first architecture that runs existing high-level interpreted languages and is demonstrably resistant to side channels.
3. We provide an implementation of DOVE for R, creating the first side-channel resistant R programming stack.
4. We evaluate the security and performance of DOVE against evaluation programs drawn from the genomics literature. Relative runtime overheads of DOVE against vanilla R on these programs range from $12.74\times$ to $341.62\times$.

2.2 Background

2.2.1 Programming in R

R is a statistical language that provides convenient interfaces for computations on arrays and matrices. Most function calls including primitive operators like addition and subtraction perform element-wise operations on array-like values.

2.2.1.1 Computation in R

R is an interpreted language [167], and its interpreter is written mostly in C and to a lesser extent Fortran and R itself. Every object is represented with an S-expression [137] such that interpreter parses R statements into S-expressions. The S-expressions are then evaluated and dispatched to the corresponding library functions written in C. Each C function runs on hardware as a compiled binary object. Thus, analyzing code written in R is more complex than analyzing code that is directly compiled and run on hardware (e.g. C, C++).

2.2.1.2 Not Applicable (NA)

R represents `null`-like, empty values with `NA`, the representation of which depends on the datatype. A real-valued S-expression in R is represented with a IEEE 754 `double`; `NA_REAL` is defined with the special double value `NaN` with a specific lower word (`1954`). The interpreter treats `NA` differently from other values, even from `NaN`. Integer and logical (i.e., boolean) S-expressions are implemented with an `int` type, so R reserves the lowest integer value `INT_MIN` for the representation of `NA_INTEGER` and `NA_LOGICAL`.

2.2.2 Microarchitectural Side-Channel Attacks

Microarchitectural (shortened as "$\mu Arch$") side-channel attacks are a class of privacy-related vulnerabilities in which a sensitive program's hardware resource usage leaks sensitive information to an adversary co-located to the same (or a nearby) physical machine [84]. Over the years, numerous hardware structures—cache architectures [157, 232, 240, 241], branch predictors [2, 74], pipeline components [7, 11, 91] and others [73, 90, 143, 163, 217, 231]—have been found to leak information in this way. Many of these attacks require that the attacker only share physical resources with the victim (e.g., Prime+Probe and the cache [130, 157] or Drama and the DRAM row buffer [163]), as opposed to sharing virtual memory with the victim (e.g. [240]).

2.2.3 Enclave Execution and Intel SGX

Enclave execution [202], such as with Intel SGX [108], protects sensitive applications from direct inspection or tampering from supervisor software. That is, the OS, hypervisor and other software are considered to be the attacker [30, 88, 94, 144, 156, 170, 183, 196, 217, 242], who will be referred to as the *SGX adversary* for the rest of the chapter. To use SGX, users partition their applications into enclaves at some interface boundary. For example, prior work has shown how to run whole applications with a LibOS [19, 46], containers [191],

and data structure abstractions [183] within enclaves. At boot, hardware uses attestation via digital signatures to verify the user's expected program and input data are loaded correctly into each enclave. Isolation mechanisms implemented in virtual memory protect enclave integrity and confidentiality during execution.

SGX uses the Enclave Page Cache (EPC) to store enclave application code and data. The EPC is stored in a protected region of memory known as Processor-Reserved Memory (PRM). The processor prevents other system components from reading the PRM with the help of another component, the Memory Encryption Engine (MEE), that provides encryption and integrity protection for the PRM [138]. The EPC has a fixed size of 64 or 128 MB, shared among all enclaves [110]. For applications requiring more memory, SGX uses an EPC paging mechanism supported by the SGX OS driver. Specifically, the OS can move pages out of/into the EPC and manipulate them as if they were regular pages from a demand-paging perspective. For security, pages moved out of/into the EPC are transparently encrypted/decrypted and integrity checked by the SGX hardware [108, 138].

2.2.3.1 Side-Channel Amplification

Despite providing strong virtual isolation, SGX enclave code is still managed by untrusted software. Prior work has shown how this exacerbates the side-channel problem described in Section 2.2.2.

First, SGX does not provide any physical isolation. Thus, nearly all of the *µArch* side-channel attacks discussed in Section 2.2.2 immediately apply in the SGX setting.

Second, importantly, the OS-level attacker has significant control over the enclave's execution and the processor hardware and thus can orchestrate finer-grain, lower-noise attacks than would otherwise be possible. For example, controlled side-channel attacks [231] and follow-on work [217] provide a zero-noise mechanism for an attacker to learn a victim's memory access pattern at page (or sometimes finer) granularity. A line of work has further shown how the attacker can effectively single-step, and even replay, the

victim to measure fine-grain information such as cache access pattern and arithmetic unit port contention [30, 88, 93, 94, 144, 196, 211].

2.2.4 Threat Model

Our goal is to prevent arbitrary non-SGX enclave software from learning anything about the users' data, other than non-sensitive information about the data such as its bit length. Given SGX's architecture, this implies protecting user data from leaking over arbitrary non-speculative *μArch* side channels (Section 2.2.2), given the powerful SGX adversary described above.We do not defend against hardware attacks such as power analysis [120], EM emissions [151], compromised manufacturing (*e.g.*, hardware trojans [233]), denial of service attacks, or speculative execution attacks [119] beyond default SGX protections.

Note, when we refer to *trusted computing base* (TCB) we mean the DOVE software that must function as intended—i.e., be free of logic bugs and control-flow hijacking vulnerabilities—for security to hold.

2.2.5 Data-Oblivious Programming

Data-oblivious (sometimes called "constant-time" in the hardware setting) programming is a way to write programs that makes program behavior independent of sensitive data, with respect to the side channels discussed in Section 2.2.2 [5, 11, 22, 23, 26, 37, 40, 53, 60, 64, 72, 78, 129, 129, 140, 145, 150, 156, 170, 184, 190, 198, 201, 207, 219, 245, 246, 248]. In the hardware setting, what constitutes data-oblivious execution depends on the intended adversary. In the SGX setting, we must assume a powerful adversary that can monitor potentially any *μArch* side channel as described in Section 2.2.3.

Thus, prior works that try to achieve data obliviousness in an SGX context [5, 72, 78, 140, 156, 170, 184, 190, 248] implement computation using only a carefully chosen subset of arithmetic operations (e.g., bitwise operations), conditional moves, branches with data-independent outcomes, jumps with non-sensitive destinations, and memory

instructions with data-independent addresses. For example, an `if` statement with a sensitive predicate is implemented as straight line code that executes both sides of the `if` and uses a data-oblivious ternary operator (such as the x86 `cmov` instruction or the CSWAP operation) to choose which result to keep.

2.3 The (Lack of) Data-Obliviousness of R

Our goal is to protect R programs (and by extension scientific computing) from the SGX adversary. As a starting point, imagine we try to run secure R code by moving the whole R stack into the SGX enclave (which is the approach taken by prior work [19, 46]). If R were data-oblivious, we could have security against the SGX adversary. However, we show that security is not guaranteed, by demonstrating subtle *μArch* side-channel attack vectors that come up in this approach.

2.3.1 Case Study

To evaluate the data obliviousness of R, we worked with an application of genomic data sharing to accurately represent the kinds of R scripts data scientists use. The specific application [13] aims to understand from genomics why honeybees from Puerto Rico are *gentle*, like European honeybees, even though they descend from *aggressive* Africanized honeybees from South America. Genomes from 30 honeybees were collected from Puerto Rico, Mexico, and the United States to provide a total of 90 genomes.

Overall, this honeybee study simulates the idea that three parties would like to derive critical information from their combined data set without the need for a trusted third party to consolidate the data. We did not work with truly sensitive data in this study, but the characteristics of the data and the data-sharing arrangement are essentially the same as would have been used in the hypothetical bipolar disorder study mentioned in the introduction. The honeybee data and code is available for download and will be a good

benchmark for future studies of secuirty for genomics.

We reproduce the study in [13] with this data using R, but truncate the total number of samples to 60 (due to machine limitations). The study relies on R code drawn from a set of 13 genetics research programs [39] that implement important statistical measurements found in the literature [83, 152, 205, 222], totaling 478 lines of R code [39]. We refer to these scripts as our *evaluation programs*. We evaluate 11 of these evaluation programs as a part of our analysis, as they operate on numeric values (rather than character strings or symbols). The functionality of each of these programs is explained below.

- `allele_sharing` A program to calculate the allele sharing distance between pairs of individuals [83].
- `EHHS` A program to calculate the EHHS values for a given chromosome [205].
- `hwe_chisq` A program to test the significance of deviation from Hardy–Weinberg Equilibrium (HWE) using Pearson's Chi-Squared test.
- `hwe_fisher` A program to test the significance of deviation from HWE using Fisher's Exact test.
- `iES` A program to calculate the iES statistics [205]. The code calls `EHHS` in computing its statistics.
- `LD` A program to calculate D, D', r, χ^2, $\chi^{2\prime}$, which are statistics based on the frequencies of alleles in the input.
- `neiFis_multispop` A program to calculate inbreeding coefficients, F_{is} [152], for each sub-population from a given set of SNP markers.
- `neiFis_onepop` A program to calculate inbreeding coefficients, F_{is} [152], for the total population from a given set of SNP markers.

- `snp_stats` A program to calculate basic stats on SNPs, including: allele frequency, minor allele frequency, and exact estimate of HWE.
- `wcFstats` A program to estimate the variance components and fixation indices [222].
- `wcFst_spop_pairs` A program to estimate F_{st} (θ) values for each pair of sub-populations [222].

We perform our analysis of the data-obliviousness of R using the code snippet in Figure 2.1 as a guiding example. This code is found in four of the 13 evaluation programs, and three more feature similar snippets. We use R version 3.4.4, compiled with default flags, on a Ubuntu Linux 18.04.4 machine for this study.

2.3.2 Example Walkthrough

`geno` is a set of samples made up of diploid Single Nucleotide Polymorphism (SNP) sequences. The database of samples is represented as an m by n matrix, where each column is one of n samples, each of which has m SNP positions. Each position in the matrix has a genotype, denoted as an integer `0`, `1`, or `2`. The sensitive data is the contents of `geno`, namely which genotype each SNP is for each sample. The matrix dimensions (m and n) are non-sensitive.

The line of code in Figure 2.1 sanitizes the input database: any entry that is not one of the three allowed genotypes is replaced with the special value `NA` (Section 2.2.1). This occurs in real data due to noise in the sequencing process; in particular, 1.5% of the SNP entries in the honeybee dataset [13] are marked as `NA`. The code first computes element-wise filters `geno != 0`, `geno != 1`, `geno != 2`, each of which produces a matrix of booleans (a mask) indicating whether the condition is satisfied for each SNP position in each sample. The logical AND (`&`) performs element-wise AND of these 3 masks (producing a new mask) which is used to conditionally assign elements in `geno` to `NA`.

Figure 2.1. R code snippet. geno is a sensitive diploid dataset.

```
1  geno[(geno!=0) & (geno!=1) & (geno!=2)] <- NA
```

Given the above code, the adversary's goal is to learn the genotype at each SNP position—that is, whether the value of each cell in geno is 0, 1, 2, or NA. Importantly, given no additional information about R's implementation, *the R-level code in Figure 2.1 follows guidelines for achieving data-obliviousness* (Section 2.2.5), which would seemingly prevent leaking the above information. For example, it applies simple arithmetic/logical operations element-wise over matrices of non-sensitive size, performs a count over a subset of samples with a non-sensitive length, etc. Thus, combining each mask with & entails performing a data-independent number of simple logical operations (&); this is traditionally regarded as safe.

Yet, this code is not data-oblivious thanks to the transformations it undergoes in the R stack before reaching hardware.

In particular, the R interpreter transforms the line of code from R into C calls. When R interprets &, it invokes the C routine given in Figure 2.2a. This snippet takes different code paths, depending on the values of x1 and x2, which the SGX adversary can detect by single-stepping [211] or by replaying the victim [196] and measuring time, branch predictor state, etc. We investigate the side-channel characteristics of this with two types of analyses: instruction-level and processor-level. The following analyses apply well-established principles for writing constant-time and data-oblivious programs (Section 2.2.5).

2.3.3 Instruction-Level Analysis

We wish to experimentally verify the presence of such side channels in the R codebase. We can identify them at the assembly instruction level, as the C code that R functions call runs as a part of a compiled library. We cover both the static analysis of individual

opcodes in the R binary, as well as dynamic analysis of execution traces of the binary for different input values.

2.3.3.1 Static Opcode Analysis

We identify the opcodes in the R binary, `libR.so`, using the `objdump` utility. This converts the compiled machine code into a human-readable opcode format. Then, we sweep over the `objdump` output, looking for vulnerable operations over data. In particular, we wish to find branches on sensitive data, which can leak control flow information and help an attacker reconstruct the secret.

Consider Figure 2.2b, which is the assembly for Lines 1 to 2 in Figure 2.2a. We note that the assembly shows a comparison (`cmp`) between the values stored in `rbp-0x58` (`x1`) and `0x0`, and `rbp-0x54` (`x2`)) and `0x0`. This constitutes a branchon sensitive data, as the code will take different paths through the code depending on the result of the computation (`je`, `jne`). In this case, the attacker learns if one of `x1` or `x2` equals 0. Since this `&` is applied to each SNP position of each sample in Figure 2.1, this information is leaked for every SNP position.

2.3.3.2 Dynamic Execution Trace Analysis

We now show how this static analysis can be leveraged at runtime to leak the secret. We use the branch-trace-store execution trace recording mechanism [109] on our Intel Core i3-6100 CPU to counts the number of instructions executed at the assembly level for different inputs. ranch-trace-store hooks in GDB allow us to step through the program, counting instructions between breakpoints. Table 2.1 for each possible input to `&`, as reported by brance-trace-store. Confirming the above explanation, we see that the instruction count equals 45 if and only if `x1` equals 0. Thus, the adversary learns whether this is the case if it can monitor a function of the instruction count. Other cases leak other pieces of information such as whether both `x1` and `x2` equal 1.

Figure 2.2. The R interpreter implementation of the **&** operator.

(a) C source code snippet of the **&** operator implementation.

```
1  if (x1 == 0 || x2 == 0)
2      pa[i] = 0;
3  else if (x1 == NA_LOGICAL || x2 == NA_LOGICAL)
4      pa[i] = NA_LOGICAL;
5  else
6      pa[i] = 1;
```

(b) The Intel-syntax x86-64 assembly for Lines 1 and 2 of the C code in Figure 2.2a, lightly edited for clarity.

```
; x1 in [rbp-0x58], x2 in [rbp-0x54]
a8: cmp   DWORD PTR [rbp-0x58],0x0 ; x1==0
ac: je    b4 ; if true, jump to pa[i]=0
ae: cmp   DWORD PTR [rbp-0x54],0x0 ; x2==0
b2: jne   cf ; if false, jump to else if
b4: mov   rax,QWORD PTR [rbp-0x50]
b8: lea   rdx,[rax*4+0x0]
c0: mov   rax,QWORD PTR [rbp-0x8]
c4: add   rax,rdx ; calc addr of pa[i]
c7: mov   DWORD PTR [rax],0x0 ; pa[i]=0
cf: ...
```

2.3.4 Intel PCM Analysis

Opcode and execution trace analysis is not sufficient to cover the diverse (and undocumented) set of potential *µArch* side channels, such as timing differences. We wish to show that the data dependent execution visible at the opcode layer can be verified by an attacker with access to side-channel information. Intel Processor Counter Monitor (PCM) is an Application Programming Interface (API) to monitor performance of Intel processors [54]. PCM offers various performance metrics, some of which are direct indicators of side-channel vulnerabilities. Such *µArch* measurements include cycle counts and L2/L3 cache hits. We leverage this API to experimentally check data-obliviousness of R function implementations.

We examined every performance metric that can be collected from PCM and chose

Table 2.1. The associated x86-64 instruction counts for different permutations of `x1` and `x2` fed as input to `&` in R.

Expression	Value	Instruction Count
`0 & 0`	`0`	45
`0 & 1`	`0`	45
`1 & 0`	`0`	47
`1 & 1`	`1`	54
`0 & NA`	`0`	45
`1 & NA`	`NA`	57
`NA & 0`	`0`	47
`NA & 1`	`NA`	53
`NA & NA`	`NA`	53

Table 2.2. Intel PCM Functions used for dynamic analysis.

Function Name	Criterion
`getCycles`	cycle counts
`getCyclesLostDueL3CacheMisses`	cycle counts, cache H/M
`getCyclesLostDueL2CacheMisses`	cycle counts, cache H/M
`getL2CacheHitRatio`	cache H/M
`getL3CacheHitRatio`	cache H/M
`getL3CacheMisses`	cache H/M
`getL2CacheMisses`	cache H/M
`getL2CacheHits`	cache H/M
`getL3CacheHitsNoSnoop`	cache H/M
`getL3CacheHitsSnoop`	cache H/M
`getL3CacheHits`	cache H/M
`getBytesReadFromMC`	bytes from/to MC
`getBytesWrittenToMC`	bytes from/to MC
`getIORequestBytesFromMC`	bytes from/to MC

metrics (listed in Table 2.2) that are relevant for $\mu Arch$ side-channel detection. These metrics cover one or more of three criteria: cycle counts, cache hits/misses and bytes from/to the memory controller. These API functions all begin with prefix `get` and are followed by the metric they measure.

We illustrate an example using one of these measurements, cycle counts. In this simple experiment, we show how such small differences in instruction count from Table 2.1 translate into measurable effects. We measure the number of cycles taken to evaluate one million iterations of expression `0 & 0` against those of `1 & 0`. Having access to a large

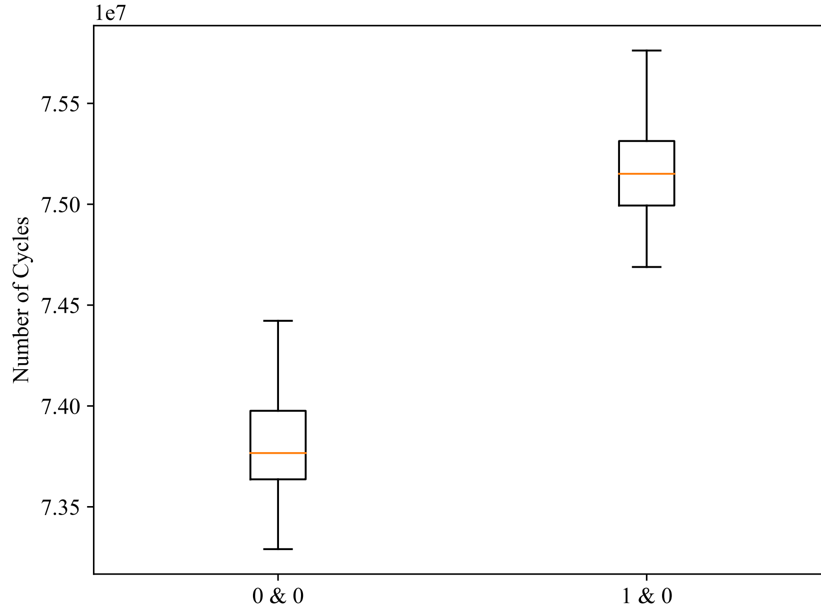


Figure 2.3. Number of cycles taken to run one million iterations of 0 & 0 and 1 & 0. Each boxplot represents 100 measurements of each expression.

number of measurements may occur naturally, *e.g.*, if the sensitive data is accessed in a loop, or if the attacker performs a $\mu Arch$ replay attack [196]. Note that the difference of execution length between two expressions is only two x86-64 instructions in Table 2.1.

Figure 2.3 visualizes 100 trials of cycle count measurements against the aforementioned two sets of inputs in boxplots. The left box shows distribution of 100 measurements for each million iterations of expression `0 & 0` and the right box represents measurements for expression `1 & 0`. On average, it took $\mu_{00} = 73.9$ million cycles ($\sigma_{00} = 441$k) for `(0 & 0)`, but it took $\mu_{10} = 75.2$ million cycles ($\sigma_{10} = 416$k) for `(1 & 0)` on average. The cycle count differences vary by a noticeable margin in the evaluation of these two expressions; even in the box plot, there is a clear separation between the experimental cycle counts of `0 & 0` and `1 & 0`.

2.3.5 Discussion

These examples are only a small subset of the parts of R that leak sensitive information. We discovered similar issues for other logical operators `|` and `xor()`, as well as functions like `sum()` found in its standard library. This, of course, does not preclude vulnerabilities arising from data-dependent R code. For example, an `if` statement with a sensitive predicate can reveal that predicate to the SGX adversary [2, 74] in R as well. Making matters worse are vulnerabilities due to timing side channels of just-in-time compilation [31], the timing differences of primitive C operations on floating point numbers [11] (such as `fdiv`, used throughout R), and the use of data-dependent `glibc` C library functions (e.g., `pow(y,x)` and `log(x)`).

R is a large code base comprising 992,564 lines of code, and is composed of hundreds of API functions and other features, implemented in a combination of R, C and Fortran [167].[2] Thus, all existing *µArch* side-channel attacks on C and Fortran applications must be considered when assessing security of R stack.

Side-channels in R present a serious security problem. Many data scientists and statisticians use R to compute on sensitive data every day. Clearly, it is not tractable for these users to understand the security implications of the code they write. At the same time, R's large code base makes manually patching data leaks inherently haphazard and error prone, even for security experts. As a result, experts have hitherto focused on replicating R's functionality in a new language/stack [190]. While these techniques add security, they trade-off expressiveness and usability by forcing data scientists to rewrite their code for a new programming stack.

In the next section, we address this challenge by designing the first secure R stack, where data scientists can program in (nearly) unchanged R, interact with the same R functionality with which they are familiar, and have strong confidence there are no latent

[2] Specifically, there are 388,141 lines of C, 345,547 lines of R and 258,876 lines of Fortran in the version of the R source we used for this work.

side channels.

2.4 The Data-Oblivious Virtual Environment

We now describe our solution to these problems, the Data-Oblivious Virtual Environment (DOVE). This begins with a discussion of our design principles and solution overview (Sections 2.4.1 and 2.4.2). Section 2.4.3 discusses the Data-Oblivious Transcript (DOT), which serves as the link between high-level programming and data-oblivious execution. Section 2.4.4 discusses the DOVE frontend, which is a set of classes that convert R code into the DOT, using pseudonyms instead of sensitive data. Finally, Section 2.4.5 describes the DOVE backend, an SGX enclave that converts the DOT operations on pseudonyms to data-oblivious computation on the actual sensitive data.

2.4.1 Design Principles

To be a practical, yet secure, programming environment for outsourcing scientific computation, DOVE requires the following:

- *Correctness*. It is necessary to provide some evidence that computed values are correct, at least for a basic collection of computations. Importantly, R code run in DOVE must have the same output as R code run outside of DOVE.
- *Expressiveness*. It is important to demonstrate that it can code enough interesting cases to be worthwhile. DOVE should be able to handle enough R functionality to be a reasonable system for data science. Additionally, DOVE should not require any changes or modification for a user's library of data processing scripts. In other words, DOVE should be transparent to the user.
- *Data-Obliviousness*. Data-oblivious computation techniques defend computation from the SGX adversary described in Section 2.2.3. DOVE should attempt to defend

against all known $\mu Arch$ side-channels, such as the ones described in Section 2.3, but be modular enough such that it can be easily patched in case a new class of side-channel is found.

- *Efficiency*. DOVE computations must sufficiently limit computational overhead. Some overhead is to be expected due to side-channel hardening and use of SGX, but it should not be so much as to prevent real data-science applications.

We developed this set of principles as a result of our experiments on R. We wanted to combine the expressiveness of R scripts with a data-oblivious core, while maintaining the efficiency required for data science (and, of course, the correctness). Our DOVE design aligns to these goals, and we evaluate our success in achieving them in Section 2.5.

2.4.2 Overview

DOVE's security objective is to evaluate programs written in high-level (e.g., interpreted) languages in a data-oblivious manner (Section 2.2.5). The key insight is that an operation that is truly data oblivious does not require the actual data to be present. Instead, the operation can take place on a *pseudonym* of the data. These pseudonyms have the same interface as normal data of the same type and support the same operations. For example, matrices are replaced with matrix pseudonyms, and matrix pseudonyms can be computed upon using the same operations as normal matrices (e.g., element-wise addition, matrix multiplication). However, the pseudonym contains no sensitive data, i.e., all of its data entries are replaced with $\perp$. This pseudonym is constructed solely through non-sensitive information specified for each pseudonym, such as, for matrices, the number of rows and columns. However, since the pseudonym does not actually have the data, any operation on the pseudonym is functionally equivalent to a NOP, i.e., $* \oplus \perp \rightarrow \perp$ where $*$ is a wildcard for any data value and $\oplus$ is an operation on the data. Instead, the operation performed is appended to a log. This log, which we call a *Data-Oblivious Transcript (DOT)*,

is thus akin to a straight-line representation of the execution of the input program. The DOT can then be replayed on the *actual* data, executing the same operations as the input program.

With this in mind we propose the following architecture, shown in Figure 2.4. Our architecture is broken into two components, making up a *frontend* and *backend*. Each of N clients runs the same input — a common (non-sensitive) high-level program — in their local environment ("frontend"). The frontend replaces any references to sensitive data with pseudonyms and generates a DOT of the input program. Although only a single DOT needs to be generated for evaluation later on, each client can optionally compute its own DOT for program integrity-checking purposes (see Section 2.4.4 for more information). This TEE ("backend") hosts the DOVE virtual machine, which is built with data-oblivious primitives. The virtual machine checks that all DOTs are equivalent (optional, for integrity) and runs the operations listed on the actual data.

Intuition for security comprises two parts. First, because the DOT is conceptually an execution trace, the backend TEE evaluates the same operations in the same order as the R program input to the frontend, regardless of the sensitive data provided to the backend. Importantly, the DOT was not created using any sensitive data, so the functions listed in the DOT are inherently independent/oblivious of that data. Second, we will architect the backend to ensure each operation is data oblivious, using well-established techniques for constant-time/data-oblivious execution.

The above architecture is general. The frontend can be adapted for different high-level languages (e.g., R, Python, Ruby), and the backend can be implemented for a variety of TEEs (e.g., SGX, TrustZone). For the rest of the chapter, we explain, design, and evaluate ideas assuming the frontend input language is R and the TEE is SGX.

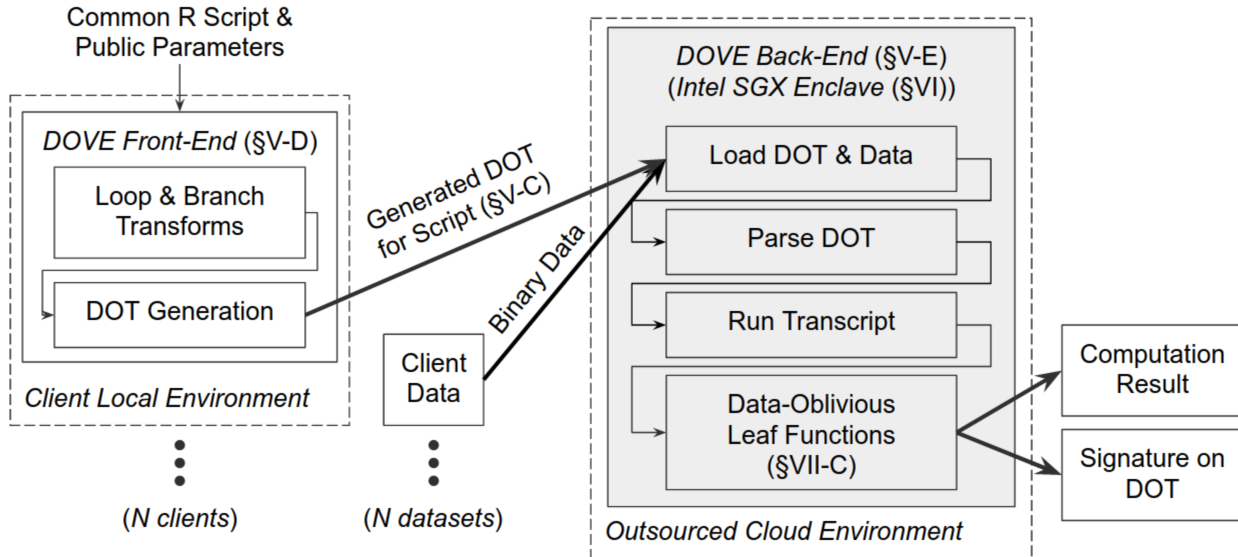


Figure 2.4. High-level overview of DOVE. Bold-face arrows between nodes represent communication over (mutually-authenticated) TLS, while thinner ones are intra-process communication within a component. Shading indicates the location of our trusted computing base (TCB).

2.4.3 Data-Oblivious Transcript (DOT)

Relevant design principles: expressiveness, data-obliviousness, efficiency.

The Data-Oblivious Transcript, or DOT, forms the core of the DOVE architecture, bridging an input program written in a high-level language with data-oblivious execution on a secure enclave. The DOT is designed to be built using only parameters related to the computation that are non-sensitive (such as data size). Because DOTs in DOVE are generated automatically, the client programmer does not need to learn the DOT language to write data-oblivious code. Once generated, the DOT is sent to the backend, where it is used to "replay" the same operations on the actual data (Section 2.4.5).

What to include in the DOT semantics strongly influences the TCB size in the backend and DOVE's overall performance. The structure of the DOT is similar to straight-line code where every operation is evaluated in the order it appears. Conditionals, data-dependent loops, etc. must be emulated with predicated, bounded execution as described below. Then, what primitive operations to include in the DOT semantics becomes a

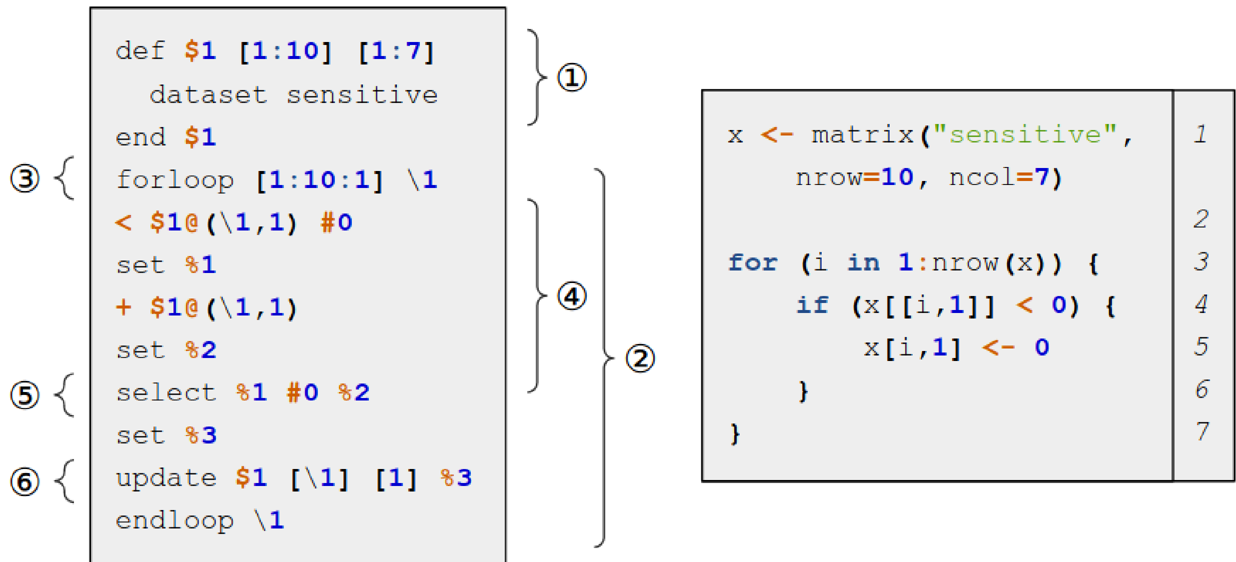


Figure 2.5. A DOT (left) and its associated R program (right). The matrix `x` corresponds to the pseudonym **$1** in the DOT, and the loop index `i` with `\1`. ① corresponds to line 1 of the program, ② the `for` loop on line 3, ③ the `if` statement on line 4, and ⑥ the assignment in line 5. Intermediate values are stored in variables marked with %, and constants are declared using #.

security/performance trade-off, because the cost to parse and run each operation in the DOT incurs non-negligible overhead in our current implementation (Section 2.5.3). For example, DOVE might implement a transcendental function such as `sin` as a single primitive operation in the DOT or as a sequence of simpler operations in the DOT (such as bitwise operations). The former design is higher performance but requires a larger TCB: the backend parses a single DOT operation and evaluates that operation using a dedicated data-oblivious implementation of `sin` in the target Instruction Set Architecture (ISA), e.g., x86-64. The latter has the opposite characteristics: the backend parses each bitwise operation yet only needs dedicated support to implement data-oblivious bitwise operations. In these situations, we decide what operations to include in the DOT semantics on a case-by-case basis, described below and in Section 2.4.4.

We now discuss DOT semantics in more detail, using Figure 2.5 as a running example. We break the discussion into two parts, first describing data creation and operations on said data, and second describing (data-oblivious) control flow. A formal EBNF grammar

Figure 2.6. The grammar for the DOT language, presented in extended Backus–Naur form. **digit** consists of 0-9, **nonzero-digit** of 1-9, and **alpha** of A-Z and of a-z.

```
loop = 'forloop' index-var '\n' {instr} 'endloop' index-var ;

instr = (def-matrix | scalar-instr | edit-instr | select-instr) '\n' ;

def-matrix = create defn end ;
create = 'def' ' ' ['const' ' '] matrix ' ' length ' ' length '\n' ;
defn = rows | dataset | matrix-instr | bind-instr ;
rows = row {row} ;
row = '\t' 'row' ' ' natural ' ' scalar {' ' scalar} '\n' ;
dataset = '\t' 'dataset' ' ' string '\n' ;
end = 'end' ' ' natural '\n';

scalar-instr = (scalar-summary-instr | ops-instr | 'set' | 'indexvar') ' ' arg
  [' ' arg] ;
scalar-summary-instr = 'any' | 'all' | 'sum' | 'prod' | 'min' | 'max' ;

matrix-instr = (ops-instr | 'empty' | 'rand' | '%*%') ' ' arg [' ' arg] ;
bind-instr = ('cbind' | 'rbind') ' ' arg ' ' {arg} ;

ops-instr = arith-instr | compare-instr | is-instr | logic-instr | math-instr
  ;
arith-instr = '+' | '-' | '*' | '/' | '^' | '%%' | '%/%' ;
compare-instr = '==' | '<=' | '>=' | '>' | '<' | '!=' ;
is-instr = 'NA?' | 'INF?' | 'NAN?' ;
logic-instr = '!' | '|' | '&' ;
math-instr = 'abs' | 'sign' | 'sqrt' | 'floor' | 'ceiling' | 'exp' | 'log' |
  'cos' | 'sin' | 'tan' ;

edit-instr = ('update' | 'slice' | 'slice const' | 'dim') ' ' matrix ' ' seq '
  ' seq ' ' matrix ;
select-instr = 'select' ' ' arg ' ' arg ' ' arg ;

length = '[1:' natural ']' ;
seq = ordered-seq | unordered-seq ;
ordered-seq = '[' integer ':' integer ':' integer ']' ;
unordered-seq = '[' integer {',' integer} ']' ;

arg = matrix | scalar ;
matrix = '$' natural ;
scalar = pointer | register | value | loop-index;
pointer = '$' natural '@' '(' integer ',' integer ')' ;
register = '%' natural ;
loop-index = '\' natural ;
value = '#' (float | 'NaN') ;

integer = '0' | ['-'] natural ;
natural = nonzero-digit {digit} ;
string = {digit | alpha} ;
float =  digit {digit} '.' digit {digit} ;
```

for the DOT can be found in Figure 2.6.

2.4.3.1 Data Creation, Types and Operations

We first discuss variable declarations, types and primitive operations.

Data types When the frontend transcribes a program into a DOT, the DOT grammar only allows program inputs to be (1) fixed, concrete values or (2) pseudonyms. The two basic types of pseudonyms are matrices and scalars, with matrices being composed of $m \times n$ scalar (i.e., numeric) elements. Each operation on a matrix is usually decomposed into an operation on (1) its rows, (2) its columns or (3) its elements. Thus, in the case where matrix dimensions are non-sensitive, the sequence of operations needed to compute on actual matrix data is fully captured in the DOT.

Operations on data Core functions comprise the set of primitive operations available to the DOT, including mathematical and logical operators (e.g. `+`, `==`), common mathematical functions (e.g. `exp`, `sin`), and summary operations (e.g. `sum`, `prod`).

There are two flavors of operations supported in the DOT, shown in first two rows of Table 2.3. The Safe DOT/Core category contains operations deemed safe to operate on sensitive data in the backend. Every operation in this set must be implemented data-obliviously by a compliant backend, i.e., its evaluation must result in operand-independent resource usage on the target microarchitecture (see Section 2.2.5). Each operation in this set has the following type signature: *if at least one operand is a pseudonym, the result is a pseudonym.* This is similar to taint algebras in information flow [179, 203] where if one operand is tainted, the result is tainted.

The Unsafe DOT/Core category contains operations which the DOT deems not safe to operate on sensitive data. For example, the `forloop` construct. These operations are only allowed to take non-pseudonyms as operands. Importantly, the selection which

operations are marked Unsafe is a design choice. An alternate set of DOVE semantics can specify a Safe variant of any Unsafe operation, subject to the constraint that the backend must support a data-oblivious implementation of said Safe operation.

To summarize, we have:

- *Rule 1.* If an operation's operand(s) are pseudonyms, the result is a pseudonym.
- *Rule 2.* Safe operations may take pseudonyms or non-pseudonyms as inputs. Safe operations must be implemented data obliviously by the DOVE backend.
- *Rule 3.* Unsafe operations may only take non-pseudonyms as inputs.

This is analogous to the Data-Oblivious ISA policy *Confidential data↛Unsafe instruction*, which is analogous to the classic policy *High↛Low* in information flow. If a DOT follows the above rules, we call it a *valid DOT*. Whether a DOT is valid is checked before the DOT is evaluated by the backend (Section 2.4.5), and invalid DOTs are disallowed.

2.4.3.2 Control Flow

For reasons discussed above, the DOT disallows traditional control-flow constructs such as **`if`**, **`while`**, and **`goto`**, but supports predicated execution and bounded-iteration loops (similar to the program counter model [145]).

Bounded iteration The DOT provides a `forloop` iteration primitive that only allows non-sensitive/non-pseudonym predicates. This primitive further does not support infinite loops. Loop indices are declared as non-pseudonyms. We note that supporting `forloop` is purely a performance/DOT size optimization. Equivalently, the loop could have been unrolled and the `forloop` construct removed.

Predicated conditionals The DOT supports a `select` primitive that takes a pseudonym-typed predicate and returns one of two pseudonym operands based on the value of the

predicate. `select` supports both scalar (i.e., logical `0` and `1`) and matrix predicates. Matrix predicates are transformed into element-wise select operations between the predicate and result/operand matrices. Thus, the predicate and its operands must have the same dimensions.

2.4.4 Frontend

Relevant design principles: correctness, expressiveness, efficiency.

The frontend takes R program with non-sensitive parameters as input and outputs a DOT. We develop our prototype frontend for R, but stress that the structure of the DOT is language-agnostic. As in a traditional compiler stack, one could design a different frontend for a different language that likewise compiles into the a DOT.

Before initialization, clients share non-sensitive information, such as names and dimensions of datasets, with each other. The data within each dataset is considered sensitive and is not shared. To create a DOT, a client sources the DOVE frontend, which loads the names and dimensions for each sensitive input and creates a pseudonym for each in the R environment. The client then runs their program, performing operations as normal. Instrumentation in the R interpreter (see below) records each operation into the DOT, translating each dataset to primitives supported by the DOT semantics (e.g., scalar and matrix types). Clients can access elements, assign new values, apply operators, and run functions, all while dealing only with pseudonyms. Because the frontend does not have the actual data, this transcription is data-oblivious by design.

Table 2.3. DOVE functions/operations. Functions in group "DOT/Core" are implemented directly in the DOVE backend and are included in the DOT semantics. Functions in the group "Supplemental" are implemented using operations in "DOT/Core" and exposed to the user as library functions. Safe functions require a data-oblivious implementation in the backend as they may receive pseudonyms as operands. Unsafe functions do not require a data-oblivious implementation, but can only take non-pseudonyms (non-sensitive) data as operands.

Group	Functions					
Safe DOT/Core (in TCB)	abs	sqrt	floor	ceiling	exp	log	cos
	sin	tan	sign	+	-	*	/
	^	%%	%/%	>	<	>=	<=
	==	!=	|	&	!	all	any
	sum	prod	min	max	range	is.na	is.nan
	is.infinite	select	%*%	cbind	rbind		
Unsafe DOT/Core (in TCB)	forloop	dim	[	[[			
Supplemental (not in TCB)	fisher.test	pchisq	mean	colMeans	colSums	rowMeans	rowSums
	is.finite	as.numeric	as.matrix	apply	lapply	unlist	which
	data.frame	matrix	split	pmin	pmax	nrow	ncol
	len	t					

Our DOVE implementation ensures interface compatibility with base R in the implemented functions of the frontend. We use R's S3 method dispatch to overload functions in base R for pseudonyms. This requires no modification to the R interpreter, as clients merely have to import the DOVE frontend in their existing programs; in most cases, no programmer intervention is necessary.

Table 2.3 lists all functions available to programmers. The Safe and Unsafe "DOT/Core" group of functions are those included in the DOT semantics (see previous section). To provide a richer library for clients, we also provide a "Supplemental" group of functions which are built using only the operations in "DOT/Core". For example, `colSums` calls the DOT function `sum` in a loop over the columns of a matrix. We provide these functions to enhance the user programming experience and to show that our DOT functions are sufficient primitives to develop more complex functions. Note that the "Supplemental" functions do not add to size of the TCB. They do not require changes to DOT semantics and therefore do not change the backend implementation.

2.4.4.1 Construct-Specific Handling

We now describe how the frontend translates different R programming constructs to the DOT semantics from Section 2.4.3.

Bounded iteration Native R's `for` loop is not DOT-aware, so it just repeats the body of the loop m times. Instead, the frontend automatically transforms such bounded loops to use the `forloop` DOT construct. In our testing, we observed a $>99\%$ decrease in frontend runtime using the DOT's `forloop` loops over normal `for` loops for compute-heavy $O(m^2)$-complexity programs. Early loop termination (e.g., `break`) is transformed in a manner similar to those of prior works [37, 129].

Predicated conditionals The frontend must translate conventional if-then-else structures into the predicated execution model supported by the DOT (Section 2.4.3). For this, we implement an if-conversion transformation that is similar to prior works [53, 170]: an if-else with a sensitive predicate is converted into straight-line code where both sides of the if-else are unconditionally evaluated and a DOT `select` operator is used to choose the correct results at the end. Our frontend automatically converts R `if` statements to use the `select` primitive (discussed in Section 2.4.3) in the DOT. The whole expression is then recorded into the DOT directly; since the frontend does not have access to the actual data, the DOT must necessarily record both sides of the condition.

Disallowed constructs Overall, the frontend's job is to translate R semantics into DOT semantics. Sometimes this is not possible, in which case the frontend signals an error. We explain two such cases (which are also common issues in related work). First, the frontend does not allow loops where the predicate depends on a pseudonym. Second, the frontend does not allow running operations with unimplemented types e.g., string-based computation or symbol-based computation. For example, one genomic evaluation program named `geno_to_allelecnt` in Section 2.3.1 receives a matrix of characters as a sensitive input. This program calls string operations like substring search or string concatenation.

Importantly, mentioned before, the frontend may contain a bug that results in an invalid DOT that contains an illegal construct such as those mentioned above. Such non-compliant DOTs are checked at parse time in the backend and rejected before being run.

2.4.5 Backend

Relevant design principles: correctness, data-obliviousness, efficiency.

The backend is a trusted SGX enclave (optionally, with attestation support) that runs

the DOVE virtual machine that parses the DOT and runs the instructions contained within on the clients' sensitive data. Code in the backend ensures that only valid DOTs are run (Section 2.4.3), and includes implementations of all operations in the DOT semantics, i.e, those listed under Safe and Unsafe "DOT/Core" in Table 2.3. Each client securely uploads (e.g., over TLS) the DOT of their R program. All clients additionally upload their shares of the sensitive dataset to the backend as well, in preparation for processing, as shown in Figure 2.4.

The scope of DOVE is to block all non-speculative $\mu Arch$ side channels (Section 2.2.4). For this purpose, the backend provides a data-oblivious implementation for operations in Safe "DOT/Core" of Table 2.3. To implement these operations, we rely on a subset of the x86-64 ISA and well-established coding practices [53] for implementing constant-time/data-oblivious functions (see Section 2.5.2 for details). For example, we implement the `select` operation using the x86-64 `cmov` instruction, and all floating-point arithmetic functions are implemented using libfixedtimefixedpoint (libFTFP), a constant-time fixed-point arithmetic library created as a work-around for timing issues on floating-point hardware [11].

Importantly, what hardware operations (e.g., machine instructions) open $\mu Arch$ side channels depends on the $\mu Arch$. For example, two x86-64 processors can implement `cmov` differently: one in a safe way, one in an unsafe way (e.g., by microcoding the `cmov` into a branch plus a move [242]). DOVE is robust to new leakages found in specific $\mu Arch$ because to block a newly discovered leakage, it is sufficient to make a backend change. For example, if a vulnerability is found in `cmov`, the backend can opt to implement the DOT `select` operation using a CSWAP (bitwise operations) or other constructs.

2.5 Experimental Evaluation

We evaluate DOVE by designing experiments to validate its four design principles of correctness, expressiveness, data-obliviousness, and efficiency. We aim to use R as a baseline, comparing its results to those of DOVE. In this way, we validate DOVE using R as a reference. Our evaluations were performed on a machine with an Intel Skylake Core i3-6100 CPU, 1 TB HDD, and 24 GB of RAM, of which 19.37 GB was allocated to the SGX enclave. The machine was running Ubuntu 18.04.4 LTS and SGX software version 2.9.1 with EPC paging support. Thus DOVE's memory is not limited to EPC size, but this mechanism adds performance overhead when it is required. The frontend ran under R interpreter version 3.4.4, and the backend was compiled against `g++`, toolchain version `7.5.0-3ubuntu1~18.04`.

2.5.1 Correctness and Expressiveness

We combine our experiments to verify the correctness and expressiveness properties of DOVE. We first perform unit tests to check that individual R functions have correct output. We then proceed to use examples to show that DOVE can express solutions to real-world problems, and does so correctly.

2.5.1.1 Unit Tests

For correctness, we confirm that what we get from DOVE is the same as what we would get from R. We perform simple unit tests. First, we ensure that frontend generate the correct DOTs, transliterating R functions into the appropriate DOT primitives. Then, we verify that the DOT primitives are processed correctly on the backend and output the expected result. Finally, we validate the end-to-end functionality of the system, checking that R output and DOVE (frontend-DOT-backend) output are equivalent.

Figure 2.7. The DOVE-compatible implementation of PageRank in R.

```
1  page_rank <- function(M) {
2      d <- 0.8
3      N <- nrow(M)
4      v <- matrix(nrow = nodes, ncol = 1, rand = TRUE)
5      norm_one <- sum(abs(v))
6      v <- v / norm_one
7      M_hat <- (M * d) + ((1-d) / N)
8      iters <- 40
9      for(i in 1:iters) {
10         v[,] <- M_hat %*% v
11     }
12     v
13 }
```

2.5.1.2 PageRank

We begin with an introductory case study on the PageRank algorithm that is used as a case study on a custom data-oblivious programming language [190]. A large proportion of this algorithm is composed of matrix multiplications, which other works choose as primary performance benchmarks [128, 170]. Our DOVE implementation of this algorithm is found in Figure 2.7. Note that Line 4 is syntactic sugar to generate a random matrix in the backend, without putting those values in the DOT.

2.5.1.3 Evaluation Scripts

We demonstrate that we can conveniently (and accurately) create DOTs from R code for our evaluation programs, as described in Section 2.3.1. Using DOVE, we were able to transform (in the frontend) and run (in the backend) 11 out of the 13 evaluation programs, totaling 326 lines of R code. The first program that we could not implement, `geno_to_allelecnt`, works on character data instead of numeric data, and as such is not supported by the current types available in the DOT. The second program, `gwas_lm`, performs a Genome-Wide Association Study (GWAS) using support in R for linear models.

We were not readily able to implement this; R provides parameters to models as a formula of symbols, not values. DOVE currently does not support this paradigm, but we believe that DOVE can be extended to do so in the future.

Ten of the remaining 11 evaluation programs were automatically transformed by the frontend into data-oblivious code. Only one program, `LD`, required manual intervention, as it was written entirely in a data-dependent style. For this program we: (1) replaced some functions that are intrinsically data-dependent with data-oblivious primitives and (2) changed lines that required sensitive data-dependent array indexing with worst-case array scans. Future implementations could alternatively use an oblivious memory, e.g., [184], to avoid such worst-case work.

2.5.2 Data-Obliviousness

It is critical that the backend is secure against *μArch* side channels. Our backend is implemented in a data-oblivious style, only using constructs that are known to the side-channel free on the x86-64 ISA. To avoid side channels that can arise due to floating point numbers, we use a previously-evaluated fixed-point library, libFTFP [11], designed to provide computation on decimal numbers in constant time.

In our backend architecture, the only place we perform computation on sensitive data is in what we term *leaf functions*. These functions are at the "leaf" of our call tree, and implements a specific DOT operation on data. Up to that point in the call tree, our backend only operates on the DOT, performing instruction fetch and setting up pointers to data for the leaf functions. Only leaf functions dereference these pointers, and then read and modify sensitive data. Verifying data-obliviousness of these functions is a crucial assessment of DOVE's security promises.

Based on the above discussion, we now scrutinize whether these leaf functions enable our security guarantee, i.e., uphold Rule 2 from Section 2.4.3. We use the same set of experiments we used in Section 2.3 to verify the data-obliviousness of DOVE. For this,

Figure 2.8. Intel-syntax assembly for `BitwiseAndOp::call`, the DOVE backend equivalent of `&` in R. This snippet has been lightly edited for clarity.

```
1  ; [snip] push current register state
2  ; initialize registers
3  mov    r13,rcx
4  mov    r12,rdi
5  mov    rbp,rdx
6  mov    rdi,rsi
7  mov    rbx,rsi
8  ; convert fixed point number to int
9  call   fixed_to_int(fixed)
10 mov    rdi,rbp
11 mov    r14d,eax
12 call   fixed_to_int(fixed)
13 and    eax,r14d ; the actual operation
14 mov    rsi,r13
15 movsx  edi,al
16 ; place `and` result into fixed
17 call   place_bool_in_fixed(int8_t, fixed*)
18 mov    rcx,r13
19 mov    rdx,rbp
20 mov    rsi,rbx
21 mov    rdi,r12
22 ; [snip] pop previous register state
23 ; supercall to data-obliviously check for NAs
24 call   BinaryOp::call(fixed, fixed, fixed*)
```

we manually disassemble and analyze every binary object file associated with DOVE functions, and verify that the subset of instructions which operate on sensitive data are instructions that do not create *μArch* side channels as a function of their operands. We also inspect the PCM characteristics of DOVE to identify any missed side channels.

2.5.2.1 Static Opcode Analysis

We disassembled the object files generated during compilation and manually looked at every function that performed an operation on sensitive data. The machine instructions that run in these functions are of relevance to the security of DOVE, since insecure

Table 2.4. All x86-64 opcodes that operate on sensitive data in the leaf functions of DOVE. Those marked with * are those not found in libFTFP.

add	and	cdqe	cmovne*	cmp	imul
lea	mov	movabs	movsd	movsx	movsxd
movzx	mul	neg	not	or	pop
push	sar	sbb	seta	setae	setbe
sete	setg	setl	setle	setne	shl
shr	sub	test	xor		

instructions may leak information about the data. A slightly truncated example of this disassembly can be found in Figure 2.8 for the backend's **&** operator used in our previous examples (e.g., Figure 2.1). Note the lack of branches on conditional data, as compared to the disassembly in Figure 2.2b. Compilation on different platforms can provide different results, so this analysis may have to be reapplied.

We first analyze the leaf function instructions that take sensitive data as operands. These instructions are shown in Table 2.4. We determined this set by inspecting instruction dependencies in the `objdump` disassembly. All but one of the opcodes in Table 2.4 is considered to be a data-oblivious instruction by libFTFP, our constant-time fixed-point arithmetic library. We refer to its authors' analysis for its security [11]. The one instruction not found in libFTFP, `cmovne`, is used for conditional moves of sensitive data in the backend. This instruction is likewise shown to be data oblivious in [170]. We further verify that the above instructions use the direct register addressing memory mode for each operand, if the value stored in the register for that operand is sensitive (which also follows standard practice for writing data-oblivious code).[3] Thus, we conclude that the machine instructions operating on sensitive data in the backend do not create *μArch* side channels.

Beyond the instructions in Table 2.4, there are other instructions in the leaf functions that *do not* operate on sensitive data. Examples include jumps to implement loops with

[3] x86-64 operands can utilize one of several flavors. For example, `rax` denotes a register file read and `[rax]` denotes a memory de-reference. The former is considered safe for use in constant-time/data-oblivious programming, while the latter creates memory-based side channels.

non-sensitive iteration counts, checks to validate dimensions on operations, sanity checks for `nullptr`, and instructions associated with implementing polymorphism. Some of these are not data oblivious (e.g., jumps), but do not impact security because they operate on non-sensitive data such as matrix dimensions.

2.5.2.2 Dynamic Execution Analysis

To further corroborate our static security analysis, we also looked at runtime instruction statistics, as we did for R in Section 2.3.3.2. We used the branch-trace-store execution trace recording [109] of the DOVE backend execution, varying the input data. We found that the sequence of non-speculative dynamic instructions executed was independent of the data passed to the backend: that is, the backend satisfies the PC model [145]. Security follows from these two analyses: (a) that the backend follows the PC model and (b) that each individual instruction that operates on sensitive data consumes operand-independent hardware resource usage (previous paragraphs).

2.5.2.3 Intel PCM

We additionally validate DOVE's data-obliviousness by repeating the PCM experiments we conducted on the R interpreter in Section 2.3.4. We performed PCM tests on every function in the backend that correspond to the Safe DOT/Core groups in Table 2.3 on 14 metrics that previously described.

For these tests we generate and compare synthetic matrices with different data distributions, i.e., pairs of D and D'. As we have seen from our instruction analyses, base R implementations handle corner cases for missing data (`NA`) and/or `0` (e.g., `log()`, `exp()`), such that varying the proportions of these two values in the input matrix results in noticeable differences in execution at the $\mu Arch$ level. Thus, we test our functions with varying amounts of these two values. The first set tests whether the function is data-oblivious against `NA` or not. This set consists of matrices with five different pro-

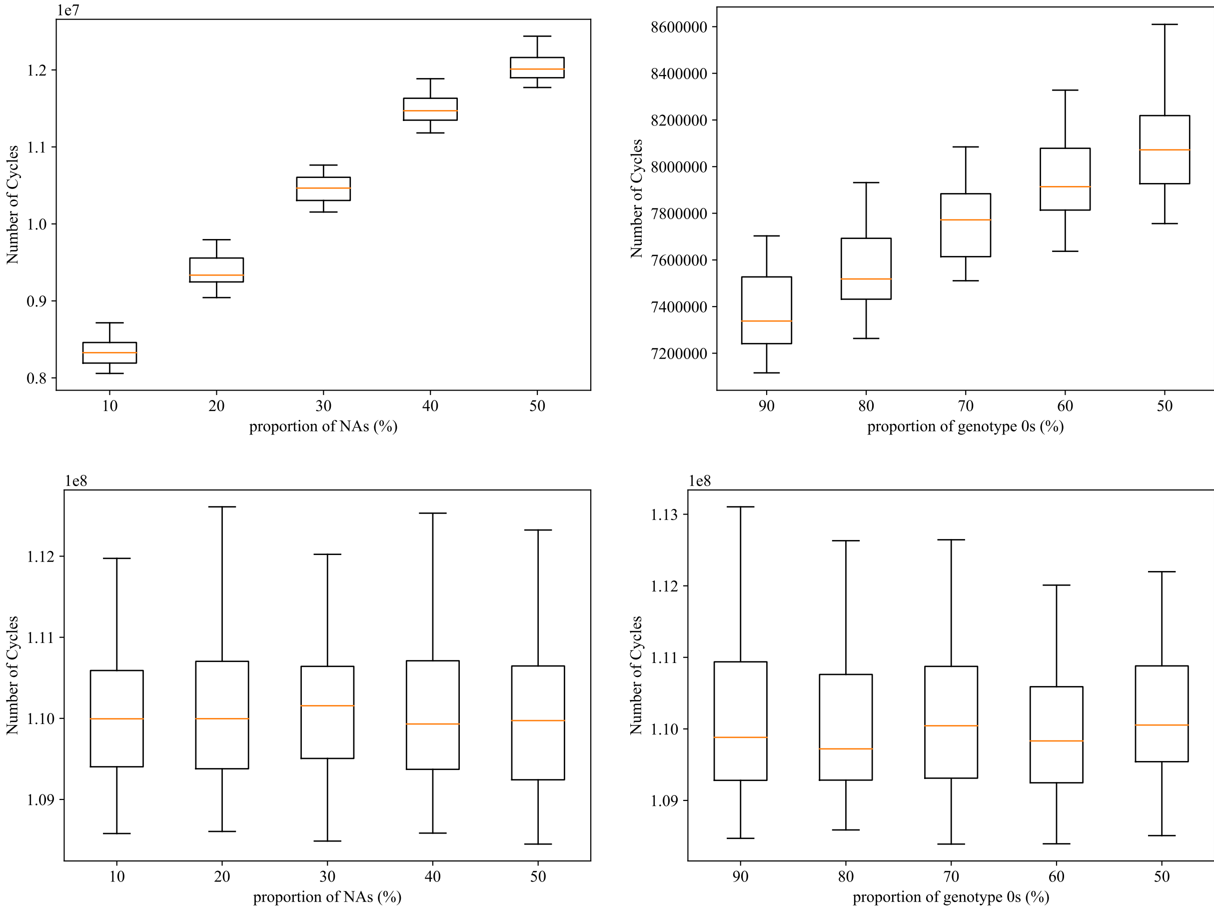


Figure 2.9. Cycle count measurements for runtime Intel PCM analysis against Line 1 of Figure 2.1. Plots above are measurements from vanilla R and plots below are from DOVE. The plots on the left are tested against varying proportions of `NA`, and plots on the right are tested against varying proportions of `0`.

portions (10%, 20%, 30%, 40%, 50%) of `NA`. A second set tests whether the function is data-oblivious against `0` or not. This set also consists of matrices with five different proportions (90%, 80%, 70%, 60%, 50%) of `0`. We generate 100 matrices of each proportion randomly in both sets for our testing. The size of each matrix is 1,000 by 60.

Figure 2.9 shows boxplots that illustrate 100 trials of cycle count measurements against the aforementioned two sets of inputs against Line 1 of Figure 2.1. Each box in the figure represents 100 measurements of random input set with varying proportions of either `NA` or `0`. When Figure 2.1 was run on vanilla R, the cycle counts differ drastically when the input's proportion of `NA` (top left) or `0` (top right) is varied. Both plots at the top shows a

linear increase in cycle counts as the proportion changes, but measurements from DOVE do not show such a trend against `NA` (bottom left) or `0` (bottom right).

2.5.3 Efficiency

We define efficiency in terms of performance, which we measure primarily through the execution time of DOVE. This is the most important metric in defining the practicality and scalability of a solution like DOVE in a data-science context.

One run of our performance benchmark is as follows. We first record the runtime of vanilla (insecure) R with data and a program. Then, we run the DOVE frontend on the same program, generating the DOT and writing it to disk. We then initialize the backend, read in the DOT, parse it, and execute the DOT instructions. Our evaluation of the DOVE implementation discusses two measures. First, we wish to consider if our frontend primitives are sufficient to express complex programs. Second, we examine the performance of DOVE when compared to its base R counterpart.

To highlight the overheads inherent to SGX and libFTFP, the external data-oblivious fixed point library [11], we ran performance benchmarks on three configurations of DOVE: (1) backend outside an SGX enclave and without libFTFP, (2) backend outside an SGX enclave and with libFTFP, and (3) backend inside an SGX enclave and with libFTFP (our default configuration). SGX-related overheads include SGX's memory encryption and access protections that isolate the enclave from the rest of the machine [56]. In particular, EPC paging (discussed in Section 2.2.3) is a significant overhead, especially for large datasets. d These overheads were exacerbated by increases in the working set of the enclave application. The libFTFP instructions' relative performance overhead is measured against its Streaming SIMD Extensions (SSE) counterpart; the overhead varies depending on the instruction, ranging from 1.2× for `neg` (operand negation) to 208× for `exp` (exponential function evaluation) [11].

We utilize the dataset from the honeybee study [13] to perform performance bench-

Table 2.5. Absolute runtimes and sizes of the evaluation programs. Programs marked with an * were run on a reduced dataset due to test system limitations. Program `iES` calls `EHHS`, so we include the lines of code from `EHHS` when measuring lines of code for `iES`. FE are measurements for frontend, NEBE are for measurements with backend without SGX, and EBE are for the backend with SGX. F indicates the use of libFTFP, the data-oblivious floating point arithmetic library that we used on our DOVE implementation. LoC stands for Lines of Code for the original R program whereas DOT size represents the size of the counterpart DOT file in bytes. Finally, the DOT overhead represents the relative overhead of the DOT's file size relative to the size of the original R program.

Program	Vanilla R (s)	FE (s)	NEBE (s)	NEBE w/ F (s)	EBE w/ F (s)	LoC (lines)	DOT size (bytes)	DOT Overhead
EHHS*	18.9	3.85	1104.43	2131.65	3575.46	40	1538	0.51
iES*	23.48	6.43	1106.34	2161.95	3625	15 + 40	159853	105.44
LD*	1787.58	3.64	2869.48	9040	32264	54	5610	0.98
allele_sharing	283.41	5.6	650.03	1841.28	29733	12	419	0.28
hwe_chisq	38.48	4.56	113.98	262.23	853.49	21	5295	4.35
hwe_fisher	690.2	4.98	141425	154194	234054	12	10287	3.92
neiFis_multispop	85.85	16.88	111.82	278.42	1077.44	38	5311	4.09
neiFis_onepop	39.13	4.9	55.85	192.53	764.38	19	7381	2.43
snp_stats	692.73	11.21	142783	155840	236644	33	1980	1.35
wcFstats	55.27	8.21	79.38	186.27	757.38	35	6624	1.58
wcFst_spop_pairs	74.05	15.43	206.55	458.26	1343.51	45	18606	5.21

marking. We run the full 2,808,570 × 60 (≈ 1.3 GB) dataset for all programs with space complexity of $O(m * n)$ where m is the number of rows and n is the number of columns. However, some of the evaluation programs could not run on this dataset due to machine limitations. Specifically, some programs with space complexity of $O(m^2)$ refuse to run even in vanilla R at full size. To address these limitations, we run a subset of programs with the first 10,000 rows of the honeybee dataset. Some related work also runs performance benchmarks on genomic data with similar sizes to that of our reduced dataset [47, 48, 180].

To normalize benchmark results run on datasets of different sizes, we present a relative overhead metric: runtime for DOVE (DOT generation, disk reading/writing, DOT evaluation) divided by runtime in vanilla R. This relative overhead metric is shown as stacked bar graphs in Figure 2.10, while raw numbers can be found in Table 2.5 Each part of the bar represents the overhead contributed by a component of the backend, categorized by three factors: the DOVE runtime's data-oblivious implementation itself, constant-time fixed point operations (libFTFP), and the use of the SGX enclave. Overall, each factor provides additional security at the cost of increased overhead. We separate our programs into two bins: programs that run on the full honeybee dataset, and programs that run on a reduced dataset due to machine limitations (marked with * across the subfigures).

The min/avg/max size overhead of each DOT relative to its R script is 0.284x/10.8x/105x. Note, the DOT may be smaller than the original program because of the DOT instruction set. We expect that the DOT can be significantly compressed. Case in point, the current DOT is represented in ASCII which is space inefficient.

We now provide more detailed analysis for several programs with noteworthy performance characteristics.

2.5.3.1 Programs with Quadratic Space Complexity

The relative overhead with DOVE is 120.7× against vanilla R on average for programs `EHHS`, `iES`, and `LD`. These three programs run statistics based on pairwise SNPs, i.e., a row is compared to each other row in the dataset. They operate in $O(m^2)$ space, or, quadratic in the number of rows m. The large relative overhead in the base DOVE implementation for `iES` and `EHHS` is due to data-oblivious transformations. Namely, the vanilla R versions of these programs benefit from early `breaks` in the loop body that occur depending on sensitive values. DOVE does not directly allow such behavior for security reasons. Hence, the backend must iterate through the entire matrix, regardless of the data, causing potentially high overhead.

2.5.3.2 Statistical Programs

The programs `hwe_chisq` and `hwe_fisher` each call a base R statistics function: `pchisq` (Chi-Square distribution) and `fisher.test` (Fisher's exact test), respectively. The program `snp_stats` calls both functions. In base R, the implementation of `fisher.test` is written in R itself whereas `pchisq` is written in C. We implement both as supplemental group functions in R Table 2.3), to provide a fair comparison and to reduce TCB size. When called, the frontend will convert the call into a series of equivalent DOT operations.

We note that, to achieve data obliviousness, our implementations of these functions are somewhat different than their vanilla R counterparts. For instance, computing a factorial of a sensitive value is intrinsically data dependent, but it is required to compute Fisher's exact test (in R, `fisher.test`). To implement factorial data obliviously, we implement it as an oblivious table lookup over a pre-determined domain of inputs, noting that other data-oblivious implementations are possible.

While `hwe_chisq` has reasonable performance overhead given our data-oblivious implementation of `pchisq`, both `hwe_fisher` and `snp_stats` show large performance

overheads. These programs call the `fisher.test` function $O(m)$ times. The insecure version of this function takes $O(n)$ time. Our data-oblivious implementation takes $O(n^2)$ time due to inefficient oblivious-memory reads. As mentioned before, a more efficient oblivious-memory primitive would reduce overhead.

2.5.3.3 Remaining Programs

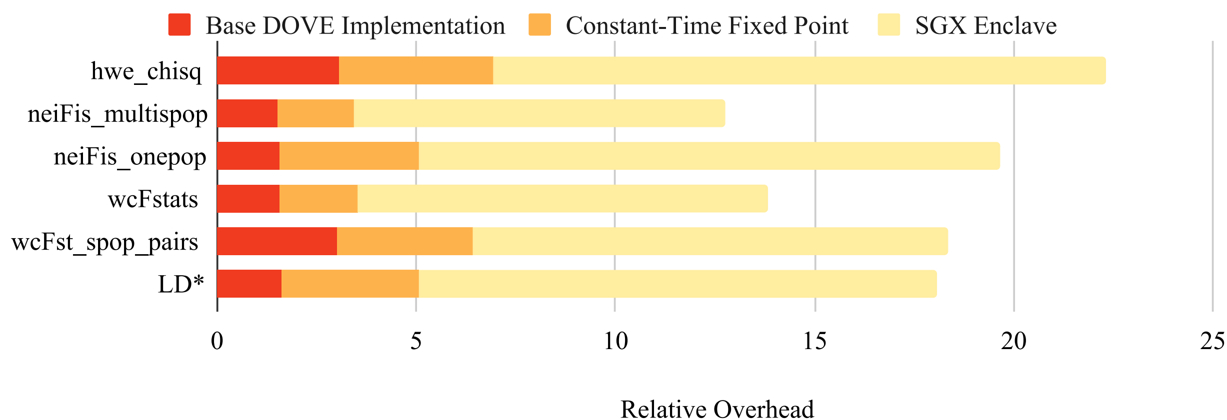
The remaining programs do not incur a significant performance penalty, as both the insecure and data-oblivious codes run in $O(m)$ time. The average overhead with DOVE is 28.3× relative to vanilla R for these programs. One program, `allele_sharing` (in Figure 2.10b), has a notably larger performance overhead than others when running inside the SGX enclave. We believe this is due to EPC paging costs. Specifically, this program has a larger working set size than SGX has EPC/PRM (2 GB vs. 64-128 MB). It further makes column-major traversals for a matrix that is stored in row-major order in memory, which leads to low spatial locality and therefore, we hypothesize, a high EPC fault rate.

2.5.4 Lessons Learned

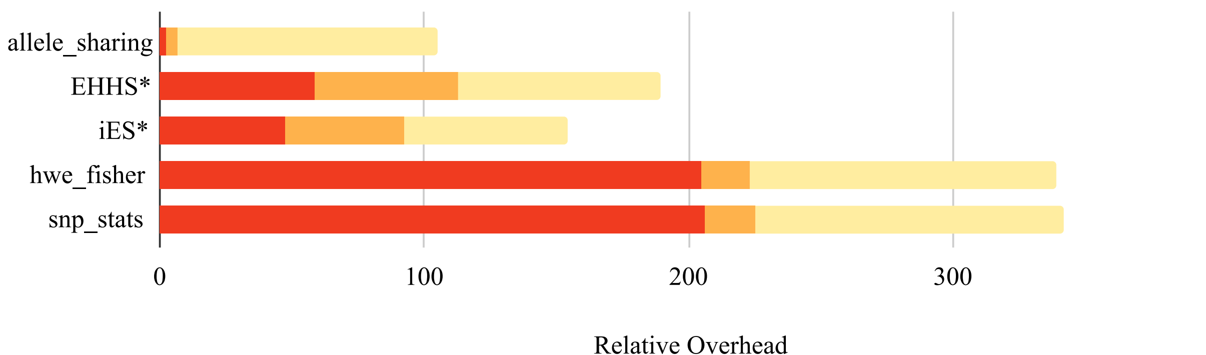
We now discuss some of the lessons we learning during the creation of DOVE. Our intermediate results helped us refine our design in three areas: expressiveness, data-obliviousness, and efficiency. We hope these observations will be useful to the community.

2.5.4.1 Automating Expressiveness

An early version of DOVE was implemented as an R library instead of directly into R's base functions. This required end-users to rewrite their code base using DOVE functions as provided by the library in order to generate a DOT. In retrospect, this was not a good design, as it restricts expressiveness to only those functions the user knows how to use with DOVE. Under this design, a user might as well learn a different, data-oblivious



(a) Programs with less than 25× relative overhead.



(b) Programs with greater than 25× relative overhead.

Figure 2.10. Performance evaluation results for the evaluation programs. Each stacked bar represents a measurement for each program. Each stack represents relative overhead of DOVE against vanilla R caused by generic data-oblivious computation, libFTFP and SGX from left to right. Programs marked with * run on reduced dataset due to machine limitations.

language. We knew that we wanted to somehow automate the transformation of R scripts into DOTs in the frontend. This is where the evaluation programs [39] were useful. These scripts contain usage of many different R constructs: S3 base functions, statistical routines, and control flow structures, among others. We used the evaluation programs as a benchmark to ensure that our automatic conversion of scripts to DOTs was expressive enough, leading to the frontend architecture in DOVE today. The DOVE frontend transparently rewrites the S-expressions for a parsed R script Section 2.2.1) to use data-oblivious primitives. The end-user, in our implementation now, does not need to manually write DOVE-compliant R code, and instead can just run scripts out-of-the-box,

thanks to the guidance provided by having a real-world evaluation set.

2.5.4.2 Data-Oblivious Statistics

R has a rich set of statistical functions baked into its standard library. To build some of this functionality into DOVE, we pulled off-the-shelf open source implementations and placed it into our backend as an external library. One of this functions is `fisher.test`, which calculates Fisher's exact test of statistical significance. Our evaluation programs [39] use this function to calculate deviation of input data from the Hardy-Weinberg Equilibrium. We did not consider the security implications of having an external library, nor did we fully understand the implementation of `fisher.test`. However, after applying data-oblivious tests to it, we noticed that it failed our instruction tests for branches on sensitive data. This is due to `fisher.test`'s use of factorials, which are data-dependent. We decided that the best way of implementing `fisher.test` was to rewrite it in data-oblivious R. The function calculates a large lookup table of factorials, and data-obliviously retrieves the correct value when needed. Thus, whenever a DOT running in the frontend calls `fisher.test`, it calls the function in the *frontend*, which itself is then transcribed into the DOT. This guarantees the security of the statistical function (and shrinks the TCB), but at a significant performance cost: 4.9× overhead for the insecure variant versus 315× for the (current) secure variant. This underscores the importance of inspecting the data-oblivious characteristics of the entire TCB.

2.5.4.3 Efficient Looping

Our original DOT design did not have a `forloop` primitive. This meant that any loops used in input R code would be fully unrolled, its instructions copied into the DOT for each loop iteration. The DOT would become size $O(n)$ for loops of size n, which we initially thought was reasonable – ostensibly, data science systems have sufficient RAM to hold a large DOT in addition to the data upon which to operate. Our experiments

showed that this was a flawed assumption. Evaluation programs that combined matrices together explicitly had reasonable performance in DOVE, while evaluation programs that explicitly looped through matrices had terrible performance. The size of the data from [13], combined with the overhead due to SGX EPC paging Section 2.5.3), slowed evaluation to a crawl. We realized that our backend had to minimize RAM in order to have reasonable performance, and we thus implemented it, reducing DOT complexity to $O(1)$ for loops of arbitrary size. We had to make additional efficiency jumps in order for our data-oblivious code to be performant, even if correctness and security properties were already guaranteed, in order to run real-world workloads.

Chapter 3

Secure Steganography for Realistic Distributions

In this chapter, we consider how to provide censorship resistance through steganography—the hiding of communication in other communication—using the latest techniques in generative modeling.

3.1 Introduction

To combat extreme censorship, there is a need for steganographic protocols that can produce *stegotext* (the steganographic equivalent of ciphertext) that closely mimics real, innocuous communication. With such techniques, it would be impossible for a censor to *selectively* repress communications, as subversive messages could hide in benign communication. For instance, if dissidents could encode secret messages into mundane appearing emails, web-forum posts, or other forms of "normal" human communication, censorship would be impractical. The ideal tool for this task is universal steganography: schemes which are able to securely hide sensitive information in arbitrary *covertext channels* (the steganographic term for communication channels). Even if the censor suspects something, the secret message cannot be found — nor is there any statistical evidence of its existence.

A key challenge in this setting is to identify a generator of some useful distribution

where sampling will produce symbols that are identical (or at least close) to ordinary content present in a communications channel. Given such a generator, numerous *universal* steganographic constructions have been proposed that can sample from this distribution to produce a stegotext [10, 14, 35, 61, 99, 194, 215]. Unfortunately, identifying useful generators is challenging, particularly for complex distributions such as natural language text. To our knowledge, the only practical attempts to achieve practical steganography such natural communication channels have come from the natural language processing (NLP) community [42, 43, 59, 75, 92, 103, 192, 214, 230, 234, 236, 243, 250]. While the resulting text is quite convincing, these works largely rely on insecure steganographic constructions that fail to achieve formal definitions [121, 139, 224, 235, 237, 238]. In this work, we focus our attention on constructing provably secure steganography for the kinds of distributions that would be difficult for a censor block without suffering significant social repercussions. To do so, we identify and overcome the barriers to using steganographic techniques as practical tools to combat network censorship.

3.1.1 Overcoming the Shortcomings of Existing Techniques

Steganographic schemes that are able to encode into any communication channel have been the subject of significant theoretical work, *e.g.*, [10, 14, 35, 61, 99, 194, 215]. Generally, constructions rely on the existence of an efficient *sampler* functionality that, on demand, outputs a *token* (sometimes referred to as a *document*) that could appear in the covertext channel. These tokens are then run through a hash function that maps the token to a small, fixed number of bits. Using rejection sampling, an encoder can find a token that maps to some specific, desired bits, usually the first few bits of a pseudo-random ciphertext. By repeatedly using this technique, a sender can encode an entire ciphertext into a series of tokens, and a receiver can recover the message by hashing the tokens and decrypting the resulting bits. Security of these approaches relies on the (pseudo-)randomness of the ciphertext and carefully controlling the bias introduced by rejection sampling.

There are two significant barriers to using universal steganographic systems for censorship-resistant communication: (1) the lack of appropriate samplers for real, desirable covertext channels, like English text, and (2) the minimum entropy bounds required to use existing techniques.

3.1.1.1 Generative Models as Steganographic Samplers

Existing work leaves samplers as an implementation detail. However, finding a suitable sampler is critical to a practical construction. Sampling is straightforward for simple covertext channels for which the instantaneous probability distribution over the next token in the channel can be measured and efficiently computed: draw random coins and use them to randomly select an output from the explicit probability distribution. Natural communication channels — the most useful targets for practical steganography — are generally too complex for such naïve sampling techniques. For example, it is infeasible to perfectly measure the distribution of the English language, and the usage of English continues to evolve and change.

Without access to perfect samplers, we explore steganographic samplers that *approximate* the target channel. While this relaxation introduces the risk that an adversary can detect a steganographic message by distinguishing between the real channel and the approximation, *this is the best we can do* when perfect samplers cannot be constructed. In this work, we propose to use *generative models* as steganographic samplers, as these models are the best technique for approximating complex distributions like text-based communication. While these models are still far from perfect, the quality of generated content is impressive [33, 168] and continues to improve, raising concerns about the disastrous societal impact of misuse [27].

Generative models operate by taking some context and model parameters and outputting an explicit probability distribution over the next token (for example, a character or a word) to follow that context. During typical use, the next token to add to the output is

randomly sampled from this explicit distribution. This process is then repeated, updating the context with the previously selected tokens, until the output is of the desired length. Model creation, or training, processes vast amounts of data to set model parameters and structure such that the resulting output distributions approximate the true distributions in the training data.

The use of generative models as steganographic samplers facilitates the creation of stegotext that are provably indistinguishable from honest model output, and thus good approximations of real communication (although not indistinguishable from real communication). We show that the nature of generative models, *i.e.* a shared (public) model and explicit probability distribution, can be leveraged to significantly increase concrete efficiency of steganographic schemes. Our key insight is that a sender and receiver can keep their models synchronized, and thus recover the *same* explicit probability distribution from which each token is selected, a departure from traditional steganographic models. This allows the receiver to make inferences about the random coins used by the sender when *sampling* each token. If the message is embedded into this randomness (in an appropriately protected manner), the receiver can use these inferences to extract the original message.

3.1.1.2 Channels with High Entropy Variability

The second barrier is the channel *entropy* requirements of most existing schemes. Specifically, most universal steganographic schemes are only capable of encoding messages into covertext channels if that channel maintains some *minimum entropy*, no matter the context. Real communication channels often encounter moments of low (or even zero) entropy, where the remaining contents of the message are fairly proscribed based on the prior context. For instance, if a sentence generated by a model trained on encyclopedia entries begins with "The largest carnivore of the Cretaceous period was the Tyranosaurus" with overwhelming probability the next token will be "Rex", and any other token would be

very unlikely. In many existing steganographic proposals, if the hash of this next token (*i.e.* Hash("Rex")) does not match the next bits of the ciphertext, no amount of rejection sampling will help the encoder find an appropriate token, forcing them to restart or abort. Thus, to ensure that the probability of this failure condition is small, most classical constructions impose impractical entropy requirements. We investigate overcoming this problem in two ways. First, we evaluate the practicality of known techniques for public-key steganography, in which an arbitrary communication channel is compiled into one with sufficient entropy. Second, we leverage the structure of generative models to create a new, symmetric key steganographic encoding scheme called Meteor. Our key observation is that the best way to adapt to variable entropy is to fluidly change the encoding rate to be proportional to the instantaneous entropy. Together, these could be used to build hybrid steganography, where the public-key scheme is used to transmit a key for a symmetric key scheme.

3.1.2 Contributions

In this work we explore the use of modern generative models as samplers for provably secure steganographic schemes. This provides the groundwork for steganography that convincingly imitates natural, human communication once the differences between generative models and true communication become imperceptible. In doing so, we have the following contributions:

Evaluation of Classical Public-Key Steganography in Practice We evaluate the use of a classical public-key steganographic scheme from [98]. We investigate adapting this scheme to work with generative models, and show that known techniques introduce prohibitively high overhead.

Meteor We present Meteor, a new symmetric-key, stateful, provably secure, steganographic system that naturally adapts to highly variable entropy. We provide formalization for the underlying techniques so that they can be easily applied to new generative models as they are developed.

Implementation and Benchmarking Additionally, we implement Meteor and evaluate its performance in multiple computing environments, including on GPU, CPU, and mobile. We focus primarily on English text as our target distribution, but also investigate protocol generation. To the best of our knowledge, our work is the first to evaluate the feasibility of a provably secure, universal steganographic using text-like covertext channels by giving concrete timing measurements.

Comparison with Informal Steganographic Work In addition to the constructive contributions above, we survey the insecure steganographic techniques present in recent work from the NLP community [42, 43, 59, 75, 92, 103, 192, 214, 230, 234, 236, 243, 250]. We discuss modeling differences and give intuition for why these protocols are not provably secure.

3.1.2.1 Deployment Scenario

Our work focuses on the following scenario: Imagine a sender (*e.g.* news website, compatriot) attempting to communicate with a receiver (*e.g.* political dissident) in the presence of a censor (*e.g.* state actor) with control over the communications network. We assume that the sender and receiver agree on any necessary key information out of band and select an appropriate (public) generative model. Although we focus on English text in this work, the generative model could be for any natural communication channel. The sender and receiver then initiate communication over an existing communication channel, using a steganographic encoder parameterized by the generative model to

select the tokens they send over the channel. The censor attempts to determine if the output of the generative model being exchanged between the sender and receiver is subversive or mundane. We note that practical deployments of these techniques would likely incorporate best practices to achieve forward secrecy, post compromise security, and asynchronicity, possibly by using parts of the Signal protocol [162].

3.1.3 Limitations

We want to be clear about the limitations of our work.

Differences Between Machine Learning Models and Human Communications Our work does not address how well a machine learning model can approximate an existing, "real" communication channel. Answering this question will be crucial for deployment and is the focus of significant, machine learning research effort [33, 168]. Regardless of the current state of generative models and how well they imitate real communication, our work is valuable for the following reasons:

1. The ever-changing and poorly defined nature of real communication channels makes sampling an inherently hard problem; channels of interest are impossible to perfectly measure and characterize. This means the imperceptibility of steganography for these channels will always be bounded by the accuracy of the available *approximation* techniques. The best approximation tool available in the existing literature is generative modeling [112], and thus we focus on integrating them into steganographic systems.

2. We prepare for a future in which encrypted and pseudorandom communications are suppressed, breaking existing tools. As such, the current inadequacies of generative models should not be seen as a limitation of our work; the quality of generative models has steadily improved [33] and is likely to continue improving. Once the

techniques we develop are necessary in practice, there is hope that generative models are sufficiently mature to produce convincingly real output.

3. Finally, there already exist applications in which sending model output is normal. For instance, artificial intelligence powered by machine learning models regularly contribute to news articles [89, 212], create art [136, 176], and create other digital content [1, 118]. Theses channels can be used to facilitate cryptographically secure steganographic communication using our techniques today.

Shared Model In Meteor, we assume that the sender and receiver (along with the censor) access the same generative model. While this requirement might seem like a limitation, we reiterate that the security of the scheme does not require that the model remain private. As such, this model is similar to the common random string model common in cryptography. Additionally, it is common practice to share high quality models publicly [33, 122, 168], and these models would outperform anything an individual could train. As such, we believe that this assumption is reasonable and show it yields significant performance gains.

3.2 Background and Related Work

3.2.1 Classical Steganography

Since Simmons' first formalization of steganographic communication [194], significant effort has been devoted to theoretical steganography. Early work focused on achieving information-theoretic security [10, 35, 141, 252] before moving on to cryptographic [14, 99, 215] and statistical [182, 197, 204] notions of steganography. The are many symmetric-key constructions [35, 99, 172], public-key constructions [14, 126, 127, 215], and even identity based constructions [177]. Relatively little on formal steganography has been in the last 15 years, although there are recent works considering the boundaries of steganography

[21], the related problem of backdoor resistance [100] and keyless steganography [3].

In general, the steganographic schemes presented in the literature rely on rejection sampling to find randomly selected elements of the covertext distribution that hash to desired bits. Given space constrains, we cannot describe and compare to all prior work. For a representative example, consider the public-key steganographic scheme from [98, 215] presented in Algorithm 3.1. First, the encoder uses a pseudorandom, public-key encryption scheme to encrypt the message. Then, one bit x_i at a time, the encoder uses rejection sampling to find a token c_i in the covertext distribution $\mathcal{D}$ such that $f(c_i) = x_i$, where f is a perfectly unbiased function over $\mathcal{D}$. We omit the formal description of the simple decoding algorithm, in which the receiver simply computes $f(c_i)$ for all i, concatenates the bits, and decrypts the result.

Security for such schemes is simple to see: each bit of the encrypted message is random, by the pseudorandomness of the cipher, and each token in the stegotext is randomly sampled from the true distribution, with no bias introduced by the hash function (by definition). As such, the distribution of the stegotext matches the covertext exactly. However, if no unbiased hash function exists, as none do for infinitely many distributions [98], a universal hash function can be used instead, and the bias it introduces must be carefully controlled.

These rejection sampling algorithms fail when the distribution has very low entropy. In such cases, it is unlikely an unbiased hash function will exist, so a universal hash function must be used. One of two possible problems is likely to occur. (1) During sampling, it is possible that the sampling bound k may be exceeded without finding an acceptable token, after which the encoder simply appends a randomly sampled token. Importantly, the receiver *can not detect* that this error has occurred, or indeed how many such errors are contained in the message, and will just get a decryption error during decoding. (2) If k is set very high, it may be possible to find a token that hashes to the correct value, at the cost of introducing noticeable bias in the output distribution. As

Algorithm 3.1: Public-Key Encoding Scheme from [98]

Input: Plaintext Message m, Distribution $\mathcal{D}$, Sampling Bound k, public-key pk
Output: Stegotext Message c

```
x ← PseudorandomPKEncrypt(pk, m)
Let x_0||x_1||...||x_|x| ← x
c ← ε
for i < |x| do
    c_i ← Sample(D)
    j ← 0
    while f(c_i) ≠ x_i and j < k do
        c_i ← Sample(D)
        j ← j + 1
    c ← c||c_i
Output c
```

Figure 3.1. The public-key steganography scheme from [98]. PseudorandomPKEncrypt is the encryption routine for a pseudorandom, public-key encryption scheme. Sample randomly selects an token from the covertext space according to the distribution $\mathcal{D}$.

such, it is critical that the distribution maintain some minimum amount of entropy. To our knowledge, only two prior works [61, 98] build stateful steganographic techniques that avoid the minimum entropy requirement. Focusing on asymptotic performance, both rely on error correcting codes and have poor practical performance.

In the closest related work, the authors of [132] theoretically analyze the limitations of using Markov Models as steganographic samplers. The prove that any sampler with limited history cannot perfectly imitate the true covertext channel. Our work overcomes this limitations by considering the output of the model the target covertext distribution.

In our work we consider more powerful machine learning models and allow the sender and receiver to share access to the same public model. This is a departure from prior steganographic work, motivated by the public availability of high quality models [33, 122, 168] and because this relaxation introduces significant efficiency gains. As there has been, to our knowledge, no work testing the practical efficiency of secure steganographic constructions for complex channels, no other work considers this model.

3.2.2 Current Steganography in Practice

The main contemporary use for steganography is to connect to Tor ([63, 171, 208]) without being flagged by the plethora of surveillance mechanisms used by censors [209]. Steganographic techniques include protocol obfuscation, *e.g.*, obfs4/ScrambleSuit [225], domain fronting [76], or mimicry, *e.g.*, SkypeMorph [142], FTEProxy [67], StegoTorus [221], CensorProofer [216], and FreeWave [101]. Although these tools allow users to circumvent censors today, they are quite brittle. For example, protocol obfuscation techniques are not cryptographically secure and rely on censors defaulting open, *i.e.*, a message should be considered innocuous when its protocol cannot be identified. Protocol mimicry techniques, encoding one protocol into another, are not always cryptographic and often fail when protocols are under-specified or change without warning [82].

Modern steganographic techniques that are cryptographically secure include tools like SkypeMorph [142], CensorProofer [216], and FreeWave [101], that tunnel information through Voice-Over-IP (VoIP) traffic, which is usually encrypted with a pseudorandom cipher. Once encrypted communication has started, a sender can replace the normal, VoIP encrypted stream with a different encrypted stream carrying the secret message. By the security of the cipher, a censor cannot detect that the contents of the encrypted channel have been replaced and the communication looks like normal, encrypted VoIP traffic. If access to encrypted or pseudorandom communication channels were suppressed, these tools would no longer work.

There have been small-scale tests [81] at deploying cryptography secure steganographic tagging via ISP level infrastructure changes, as suggested in Telex [229] and TapDance [228]. These tags indicate that a message should be redirected to another server, but stop short of hiding full messages. These tags also critically rely on the presence of (pseudo-)random fields in innocuous protocol traffic.

Practical work has been done in the field of format-transforming encryption (FTE),

such as [68, 69, 131, 154]. These approaches require senders to explicitly describe the desired covertext channel distribution, an error-prone process requiring significant manual effort and is infeasible for natural communication. None of these applications, however, provide any kind of formal steganographic guarantee. Recently, there has also been work attempting to leverage machine learning techniques to generate steganographic images, *i.e.* [16, 45, 96, 102, 187, 227], but none of these systems provide provable security.

3.2.3 Generative Neural Networks

Generative modeling aims to create new data according to some distribution using a model trained on input data from that distribution. High quality language models [33, 168], are generative neural networks, which use neural network primitives. The model itself contains a large number of "neurons" connected together in a weighted graph of "layers", which "activate" as the input is propagated through the network. Unlike traditional feed-forward neural networks used in classification tasks, generative networks maintain internal state over several inputs to generate new text. Training these models typically ingests data in an effort to set weights to neurons, such that the model's output matches the input data distribution; in other words, the network "learns" the relationships between neurons based on the input. The first practical development in this field was the creation of long short-term memory (LSTM) networks [97]. LSTM networks are found in machine translation [55, 117], speech recognition, and language modeling [112]. The transformer architecture [213], exemplified by the GPT series of models [33, 168], is also becoming popular, with results that are increasingly convincing [27].

After training, the model can be put to work. Each iteration of the model proceeds as follows: the model takes as input its previous state, or "context". As the context propagates through the network, a subset of neurons activate in each layer (based on previously trained weights), up until the "output layer". The output layer has one neuron

for output token, and uses the activated neurons to assign each token a weight between 0 and 1. The model uses its trained weights and the context input to generate a distribution of possible tokens, each with a probability assigned. The model uses random weighted sampling to select a token from this distribution, returning the chosen token as output. Finally, the returned token is appended to the context and the next iteration begins.

We note there is work focusing on differentiating machine-generated text from human-generated text [4, 15, 85]. It has yet to be seen if these techniques will remain effective as machine learning algorithms continue to improve, setting the stage for an "arms race" between generative models and distinguishers [249].

3.3 Definitions

3.3.1 Symmetric Steganography

The new construction in this work is symmetric-key stenography, so for completeness we include symmetric-key definitions. The definitions for public-key steganography are a straightforward adaptation of the definitions provided here and can be found in [98].

A symmetric steganographic scheme $\Sigma_{\mathcal{D}}$ is a triple of possibly probabilistic algorithms, $\Sigma_{\mathcal{D}} = (\mathsf{KeyGen}_{\mathcal{D}}, \mathsf{Encode}_{\mathcal{D}}, \mathsf{Decode}_{\mathcal{D}})$ parameterized by a covertext channel distribution $\mathcal{D}$.

- $\mathsf{KeyGen}_{\mathcal{D}}(1^{\lambda})$ takes arbitrary input with length λ and generates k, the key material used for the other two functionalities.
- $\mathsf{Encode}_{\mathcal{D}}(k, m, \mathcal{H})$ is a (possibly probabilistic) algorithm that takes a key k and a plaintext message m. Additionally, the algorithm can optionally take in a message history $\mathcal{H}$, which is an ordered set of covertext messages $\mathcal{H} = \{h_0, h_1, \ldots, h_{|\mathcal{H}|-1}\}$, presumably that have been sent over the channel. Encode returns a stegotext message composed of $c_i \in \mathcal{D}$.

- $\mathsf{Decode}_{\mathcal{D}}(k, c, \mathcal{H})$ is a (possibly probabilistic) algorithm that takes as input a key k and a stegotext message c and an optional ordered set of covertext messages $\mathcal{H}$. Decode returns a plaintext message m on success or the empty string ε on failure.

We use the history notation that is used in a number of previous works [99, 215], but not universally adopted. The history input to the encode and decode functions capture the notion that covertext channels may be stateful. For instance, members of the ordered set $\mathcal{H}$ could be text messages previously exchanged between two parties or the opening messages of a TCP handshake.

3.3.1.1 Correctness

A steganographic protocol must be correct, *i.e.* except with negligible probability an encoded message can be recovered using the decode algorithm. Formally, for any $k \leftarrow \mathsf{KeyGen}_{\mathcal{D}}(1^{\lambda})$,

$$\mathbf{Pr}\left[\, \mathsf{Decode}_{\mathcal{D}}(k, \mathsf{Encode}_{\mathcal{D}}(k, m, \mathcal{H}), \mathcal{H}) = m \,\right] \geq 1 - \mathsf{negl}(\breve{\ }).$$

3.3.1.2 Security

We adopt a symmetric-key analog of the security definitions for a steganographic system secure against a chosen hiddentext attacks in [215], similar to the real-or-random games used in other cryptographic notions. Intuitively, a steganographic protocol $\Sigma_{\mathcal{D}}$ is secure if all ppt. adversaries are unable to distinguish with non-negligible advantage if they have access to encoding oracle $\mathsf{Encode}_{\mathcal{D}}(k, \cdot, \cdot)$ or a random sampling oracle $O_{\mathcal{D}}(\cdot, \cdot)$ that returns a sample of the appropriate length. This ensures that an adversary wishing to block encoded messages will be forced to block innocuous messages as well. We allow the adversary to not only have a sampling oracle to the distribution (as in [99]), but also have the same distribution description given to the encoding algorithm. More formally, we write,

Definition 1. We say that a steganographic scheme $\Sigma_{\mathcal{D}}$ is secure against *chosen hiddentext attacks* if for all ppt. adversaries $\mathcal{A}_{\mathcal{D}}$, $k \leftarrow \mathsf{KeyGen}_{\mathcal{D}}(1^{\lambda})$,

$$\left|\mathbf{Pr}\left[\mathcal{A}_{\mathcal{D}}^{\mathsf{Encode}_{\mathcal{D}}(k,\cdot,\cdot))} = 1\right] - \mathbf{Pr}\left[\mathcal{A}_{\mathcal{D}}^{O_{\mathcal{D}}(\cdot,\cdot)} = 1\right]\right| < \mathsf{negl}(\breve{})$$

where $O_{\mathcal{D}}(\cdot,\cdot)$ is an oracle that randomly samples from the distribution.

3.3.2 Ranged Randomness Recoverable Sampling Scheme

To construct Meteor, we will need a very specific property that many machine learning algorithms, like generative neural networks, possess: namely, that the random coins used to sample from the distribution can be recovered with access to a description of the distribution. If it is possible to *uniquely* recover these random coins, steganography is trivial: sample covertext elements using a pseudorandom ciphertext as sampling randomness and recover this ciphertext during decoding. However, generative machine learning models do not achieve unique randomness recovery.

Meteor requires a sampling algorithm with a randomness recovery algorithm that extracts the *set* of all random values that would yield the sample. Because this set could possibly be exponentially large, we requiring that the set be made up of polynomial number[1] of continuous intervals, *i.e.* it has a polynomial space representation that can be efficiently tested for membership. We call schemes that have this property *Ranged Randomness Recoverable Sampling Schemes,* or RRRSS. The formal interface for RRRSS schemes is parameterized by an underlying distribution $\mathcal{D}$, from which samples are to be drawn and has two ppt. algorithms. Additionaly, we make the size of length of the randomness explicit by requiring all random values to be selected from $\{0,1\}^{\beta}$. The two algorithms are defined below:

- $\mathsf{Sample}_{\mathcal{D}}^{\beta}(\mathcal{H}, r) \rightarrow s$. On history $\mathcal{H}$ and randomness $r \in \{0,1\}^{\beta}$, sample an output s from its underlying distribution $\mathcal{D}$

[1] In practice, we will be working with schemes for which there is a single set, continuous set of random values that result in the same output.

- $\mathsf{Recover}^{\beta}_{\mathcal{D}}(\mathcal{H}, s) \to \mathcal{R}$. On history $\mathcal{H}$ and sample s, output a set $\mathcal{R}$ comprised of values $r \in \{0,1\}^{\beta}$

Note that our sampling scheme takes in a history, making it somewhat stateful. This allows for conditioning sampling on priors, a key property we require to ensure that Meteor is sufficiently flexible to adapt to new covertext distributions. For example, consider character-by-character text generation: the probability of the next character being "x" is significantly altered if the prior character was a "e" or a "t."

We require that these algorithms satisfy the correctness and coverage guarantees.

3.3.2.1 Correctness

We require that all of the returned randomness values would actual sample the same value. Formally, for all $r \in \{0,1\}^{\beta}$, and all history sets $\mathcal{H}$,

$$\mathbf{Pr}\left[\forall \hat{r} \in \mathcal{R}, \mathsf{Sample}^{\beta}_{\mathcal{D}}(\mathcal{H},\ \hat{r}) = s \mid \mathcal{R} \leftarrow \mathsf{Recover}^{\beta}_{\mathcal{D}}(\mathcal{H}, s); s \leftarrow \mathsf{Sample}^{\beta}_{\mathcal{D}}(\mathcal{H}, r)\right] = 1.$$

3.3.2.2 Coverage

We require that the recover algorithm must return all the possible random values that would yield the target sample. Formally, for all $r \in \{0,1\}^{\beta}$, and all history sets $\mathcal{H}$,

$$\mathbf{Pr}\left[\forall \hat{r} \in \{0,1\}^{\beta} \text{ s.t. } \mathsf{Sample}^{\beta}_{\mathcal{D}}(\mathcal{H}, \hat{r}) = s,\ \hat{r} \in \mathcal{R} \mid \mathcal{R} \leftarrow \mathsf{Recover}^{\beta}_{\mathcal{D}}(\mathcal{H}, s); s \leftarrow \mathsf{Sample}^{\beta}_{\mathcal{D}}(\mathcal{H}, r)\right] = 1.$$

We note that the structure of modern generative models trivially guarantees these sampling properties. This because all of the random values that would yield a particular output of the sample function are sequential in the lexicographical ordering of $\{0,1\}^{\beta}$.

The notion of *randomness recovery* has been widely studied in cryptography, primarily when building $\mathsf{IND-CCA2}$ secure public-key cryptography, *e.g.* [58, 161]. These works define notions like *unique randomness recovery* and *randomness recovery*, in which the recover algorithm run on some s returns a single value r such that $f(k, r) = s$ for an appropriate

function f and key k. Unlike the definitions in prior work, we require a sample scheme over a some distribution and the extraction of intervals.

3.4 Adapting Classical Steganographic Schemes

In this section, we focus on adapting classical steganographic techniques to English language distributions using generative models, specifically the GPT-2 [168] language model.

3.4.1 Characterizing Real Distributions

As noted in Section 3.2, existing steganographic schemes require a certain, minimum amount of entropy for each sampling event. Any positive value, no matter how small, is sufficient for a channel to be "always informative," *i.e.*, theoretically permit the generation of stegotext. In practice, as we will see, an always informative channel with trivial entropy will yield extraordinarily long stegotext, a problem in practice.

Practical covertext channels, on the other hand, may not be always-informative, let alone have non-trivial entropy. Figure 3.2 depicts several representative runs of the entropy over time for a sample of tokens from the GPT-2 model. Each data point reflects the amount of entropy in the model after sampling x characters from the model. The entropy varies wildly between sampling events, and there is no clear consistency state of entropy over several tokens. Moreover, the entropy occasionally drops close to zero. As such, existing steganographic techniques will fail; in our testing, Algorithm 3.1 from [98] has a 100% failure rate when encoding a 16-byte message using GPT-2.

3.4.1.1 Adaptation 1: Entropy Bounding

A natural adaptation to periods of low entropy would be to not attempt to encode information while the entropy in the channel is too low. Both the sender and receiver have access to the distribution, meaning they can both detect periods of low entropy

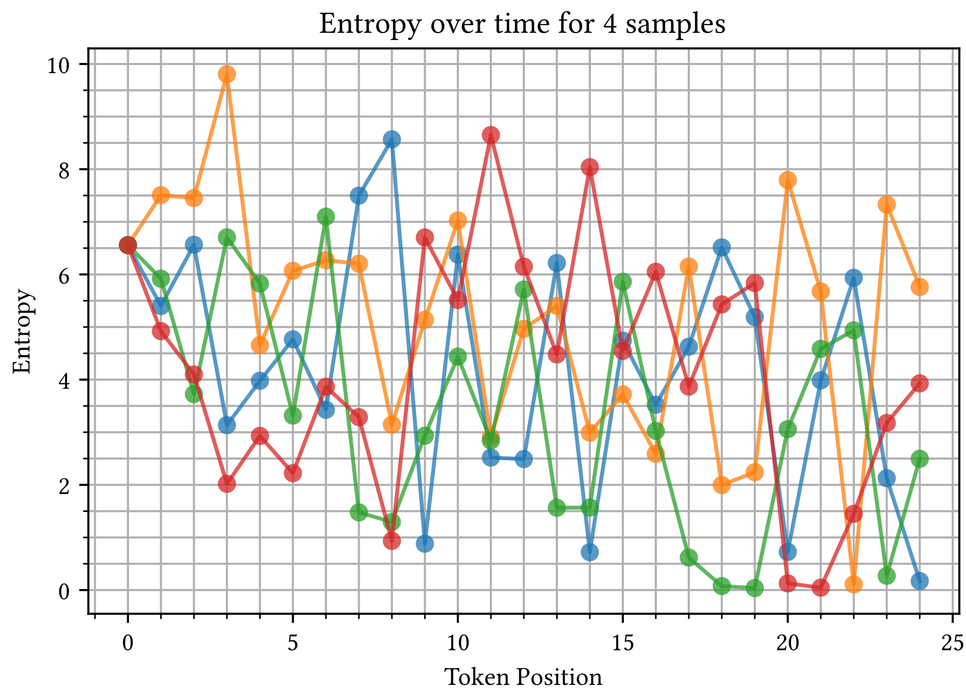


Figure 3.2. Entropy of GPT-2 output distributions. Each datapoint computed as Shannon entropy of the output distribution after sampling a certain number of tokens. Then, a random token is sampled from that distribution and appended to the context. Different colors represent different runs starting with the same context and different randomness.

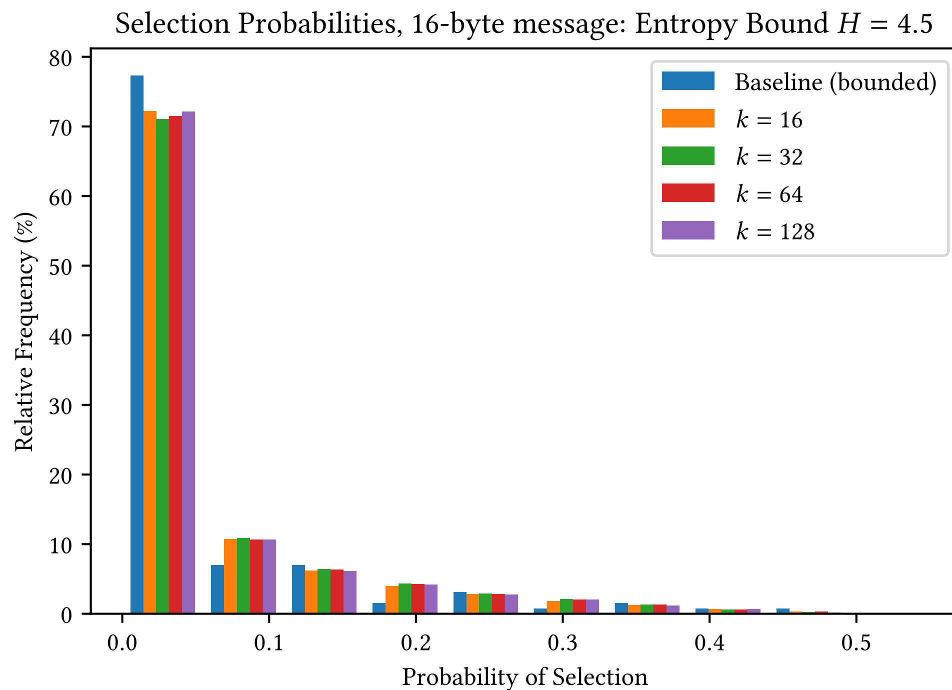


Figure 3.3. Binned probability of selecting the tokens included in the final stegotext using entropy bounding with a value of 4.5 and the GPT-2 model. The stegotext tokens clearly come from a different distribution. Note that baseline tokens were only sampled from events above the entropy bound.

and skip them. This means that only "high-entropy" events are utilized for sampling, fixing a minimum entropy that is used in the steganographic protocol. In effect, this entropy bounding creates a sort of channel-within-a-channel that meets the always entropy requirement.

While this does increase the success rate (this method achieved 0–10% failure rate in our tests), it also introduces a new problem: significant bias in the sampled tokens. Figure 3.3 is a histogram showing the probability that the selected token from the distribution would be sampled (*i.e.* the probability weight of the selected tokens). In the figure, entropy bounding for different numbers of tries are shown (k), along with a baseline sample. The baseline is also "bounded" here: it represents the probabilities of normally-sampled tokens when the distribution entropy was above $H = 4.5$.

As the figure depicts, the entropy bounding method introduces significant bias by including a disproportionate number of tokens in certain bins. This is because the hash function used is not unbiased, so repeated rejection sampling from the same distribution exacerbates the bias of the hash function. In short, there is still not *enough* entropy to hide the bias introduced by the hash function. Thus, an adversary can distinguish between an encoded message and an innocuous one by seeing if the selection probabilities of the messages are different.

3.4.1.2 Adaptation 2: Variable Length Samples

An alternative method to handle low-entropy periods, as proposed by [98], is to compile the channel into one with sufficient entropy. If a channel is always informative, meaning it always has some $\epsilon > 0$ entropy, this can be done by sampling some fixed number ℓ elements together, such that the resulting channel has at least $\ell \times \epsilon$ entropy. By setting ℓ appropriately, the entropy in the compiled channel is guaranteed to be high enough. However, in real communications channels, the entropy in the channel may not always be non-zero. As such, a naïve application of this approach will fall short.

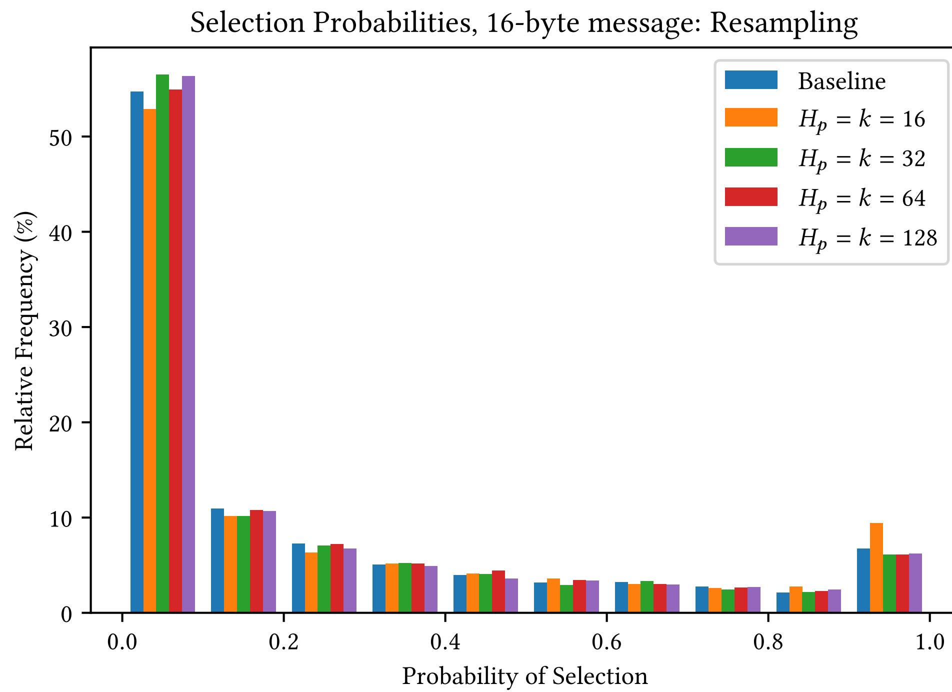


Figure 3.4. Binned probability of selecting the tokens included in the final stegotext variable length sampling. Although there is slight variation in the distributions, there is no clear difference between the stegotext and the baseline. Moreover, this method is proved secure in [98].

We overcome this by sampling a *variable* number of tokens in each sampling event, such that the cumulative entropy of the distributions from which the tokens come surpasses the minimum requirement. More specifically, instead of sampling one token at a time in the `while` loop of Algorithm 3.1, this method samples p tokens until the sum of the entropy of the distributions from which those tokens were sampled meets a minimum threshold H_p. Intuitively, this approach "collects" entropy before attempting to encode into it, boosting success rate while avoiding the issues of low entropy.

Figure 3.4 shows a selection probabilities graph, with different values of H_p compared against a baseline measurement of normal sampling from the GPT-2 (note this baseline includes all sampled tokens, unlike in Figure 3.3). In the figure, each set of runs of the model sets $\lambda = k$, i.e., the entropy required to encode is equivalent to the number of tries to encode. There are differences between the probabilities, but here is no clear pattern – this variation can be attributed to sampling error. [98] proved that for this approach to be secure, H_p must be strictly larger than $\log(k)$; to achieve useful security parameters, we need $H_p = k \approx 2 \times \lambda$, where λ is the security parameter.

While provably secure, variable length sampling results in unreasonably large stegotext and long encoding times. Table 3.1 shows the length of stegotext and encoding times when encoding a 16 byte plaintext message using adaptation 2 on our Desktop/GPU test environment using the GPT-2 model (refer to Section 3.6 for hardware details). Each row corresponds to 30 runs of the model for that set of parameters. As H_p (and thereby k) increase, the length of the stegotext also increases: the higher resampling entropy requirement means that more tokens must be sampled, which takes more time. We note that these results include GPU acceleration, so there is little room for performance boosts from hardware.

Table 3.1. Performance results for model load encoding using the method of [98] and resampling, averaged over 30 runs. The message being encoded is the first 16 bytes of Lorem Ipsum.

Parameters	Samples (Tokens)	Time (Sec)	Stegotext Len. (KiB)	Overhead (Length)
$H_p = k = 16$	502.8	42.69	2.3	149.4x
$H_p = k = 32$	880.4	128.41	4.1	261.8x
$H_p = k = 64$	1645.0	361.28	7.5	482.1x
$H_p = k = 128$	2994.6	765.40	13.6	870.7x

3.5 More Efficient Symmetric-Key Steganography

We now design Meteor, a symmetric-key steganographic scheme that is more practical than the techniques above. A more efficient symmetric-key approach would allow for hybrid steganography, in which a sender encodes a symmetric key using the public-key steganography and then switches to a faster and more efficient encoding scheme using this symmetric key. We note that while symmetric-key approaches have been considered in the past, *e.g.* [99, 172], they also rely on the entropy gathering techniques highlighted above. Our approach's intuition to accommodate high entropy variability is to fluidly change the encoding rate with the instantaneous entropy in the channel. As will become clear, Meteor does this *implicitly*, by having the *expected* number of bits encoded be proportional to the entropy.

3.5.1 Intuition

Suppose we have, for example, a generative model $\mathcal{M}$ trained to output English text word-by-word. Each iteration takes as input all previously generated words $\mathcal{H}$ and outputs a probability distribution $\mathcal{P}$ for the next word, defined over all known words $\mathcal{T}$. This is done by partitioning the probability space between 0 and 1 (represented at some fixed precision) into continuous intervals $r_0, r_1, \ldots, r_m$ corresponding to each valid word. For instance, if the precision is 5 bits, r_0 might be interval $[00000, 00101)$, r_1 might be $[00101, 10000)$, and

so on. The algorithm then generates a uniform random value $r \in [00000, 11111]$, finds the interval r_i into which r falls, and outputs the corresponding word. In the example, if $r = 01110$, then the word corresponding to r_1 would be chosen. In practice, these values all have much higher precision, for example $r \in \{0,1\}^{32}, r_i \in \{0,1\}^{32} \times \{0,1\}^{32}$.

Meteor embeds messages into the random number r used to sample from the model, as illustrated in Figure 3.5. Consider the information that a potential receiver with access to the model might learn from a single output of the generative model. Because the receiver has access to $\mathcal{M}$, they can recover the interval r_i into which r must have fallen. Note that a r_i might contain a huge — possibly exponential — number of possible values that would all yield the same sample, meaning the receiver cannot uniquely recover the true value of r. However, because the intervals are *continuous*, all such values may share a prefix, effectively fixing the first few bits of r in the view of the receiver. In this example above, all values in r_1 are contained in the first half of the distribution space, so the receiver can conclude the first bit of r must have been a 0. Similarly, if the word corresponding to r_0 had been chosen, the first bits of r must have been 00. Another example can be seen in Figure 3.5, in which the interval corresponding to the word "The" shares the prefix 01, so a receiver can recover these bits. In this way, if r is a function of the hidden message, the receiver can potentially recover bits of information about the message with each output of the model. Because the sender and receiver share the description of the distribution, the sender can determine how many bits will be recoverable, and then discard those bits before repeating the process.

The key challenge in this setting is keeping the message hidden from the adversarial censor with access to the same distribution. Clearly, using the bits of the message as the randomness is insecure, as a censor with the same model could extract the message. Encrypting the message with a pseudorandom cipher, as in the public-key solution above, is also insufficient because it is possible that the encoder will be forced to *reuse randomness*. For example, consider a probability distribution in which the values of the

interval containing r have no shared prefix, but 90% of the values in that interval begin with a 0. Because no bits are transmitted and the next iteration will use the same value of r. The censor now knows that with 90% likelihood, r in the second sampling event begins with zero. Over enough trials, a censor could detect this bias and distinguish between honestly sampled output and stegotext.

To avoid the reuse of randomness, Meteor generates a fresh mask for r each time the sender samples an output. This is done using a PRG, keyed with state shared by the sender and receiver, and applied using XOR. The receiver recovers as many bits of r as possible and then unmasks them with the corresponding XOR mask to recover bits of the message. Conceptually, this can be seen as repeatedly encrypting the message with a stream cipher, facilitating bit-by-bit decryption. This novel encoding technique means the number of bits that can be transmitted in each sampling event is not fixed. In practice, this is a huge advantage, as the expected number of bits transmitted is proportional to the entropy in the channel without requiring any explicit signaling (see Section 3.5.2.4). Finally, it is intuitively clear why this approach yields a secure scheme: (1) each sampling event is performed with a value of r that appears independent and random and (2) all bits that can be recovered are obscured with a one-time pad.

3.5.2 Meteor

For notation, let λ be a security parameter, ϵ be the empty string, and $\|$ represent concatenation or appending to an ordered set, as appropriate. We adopt Python-like array indexing, in which $x[a:b]$ includes the elements of x starting with a and ending with b, exclusive. Finally, we use two subroutines $\mathsf{LenPrefix}^{\beta}(\cdot)$ and $\mathsf{Prefix}^{\beta}(\cdot)$, presented in Algorithm 3.2 and Algorithm 3.3, respectively. The first gives the length of the longest shared bit prefix of elements in the set, and the second returns this bit prefix explicitly.

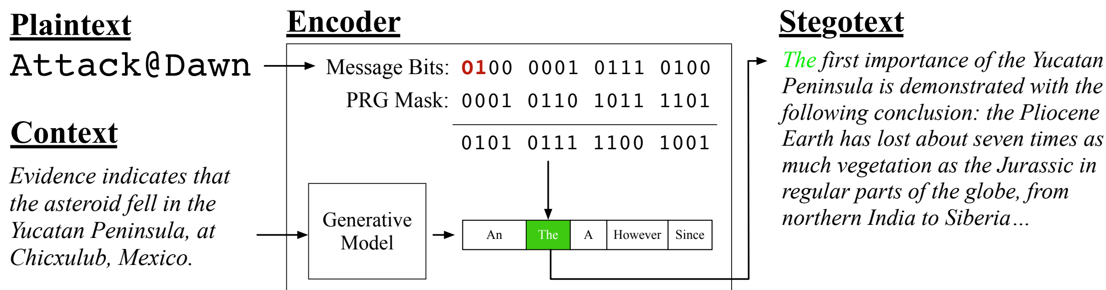


Figure 3.5. An overview of the encoding strategy for Meteor. In each iteration of Meteor, a new token (shown in green) is selected from the probability distribution created by the generative model. Depending on the token selected, a few bits (shown in red) can be recovered by the receiver. The stegotext above is real output from the GPT-2 model.

Pseudorandom Generators Our construction leverages a pseudorandom generator PRG [28]. For a more formal treatment of the security notions of PRGs, see [178] and the citations contained therein. We adopt the notation used in stateful PRGs. Specifically, let the PRG have the functionalities $\mathsf{PRG.Setup}$ and $\mathsf{PRG.Next}$. The setup algorithm generates the secret state material, which we will denote k_{prg} for simplicity, and the next algorithm generates β pseudorandom bits. We require that the PRG satisfy at least the real-or-random security games.

3.5.2.1 Construction

Meteor consists of three algorithms, parameterized by a bit precision β and a model $\mathcal{M}$ that supports a RRRSS. We use a generative model $\mathcal{M}$ as our instantiation of the distribution $\mathcal{D}$ for an RRRSS as defined in Section 3.3. The key generation algorithm $\mathsf{KeyGen}^{\beta}_{\mathcal{M}}$ is presented in Algorithm 3.4, the encoding algorithm $\mathsf{Encode}^{\beta}_{\mathcal{M}}$ is presented in Algorithm 3.5, and the decoding algorithm $\mathsf{Decode}^{\beta}_{\mathcal{M}}$ is presented in Algorithm 3.6.

The precision $\beta \in \mathbb{Z}, \beta > 0$ controls the maximum number of bits that can be encoded in each iteration. β should be the accuracy of the underlying sampling scheme. Most models in our implementation give probability distributions accurate to 32 bits, so we set $\beta = 32$. In our tests, it is incredibly unlikely that 32 bits will successfully be encoded at

Algorithm 3.2: $\mathsf{LenPrefix}^{\beta}$

Input: Set of Bit Strings $\mathcal{R} = \{r_1, r_2, \ldots r_n\}$
Output: Length ℓ
$\ell \leftarrow 1$
while $\ell < \beta$ **do**
  if $\exists\, i, j \in \{1, \ldots, n\}$ *such that* $r_i[0:\ell] \neq r_j[0:\ell]$ **then**
    Output $\ell - 1$
  $\ell \leftarrow \ell + 1$
Output ℓ

Algorithm 3.3: Prefix^{β}

Input: Set of Bit Strings $\mathcal{R} = \{r_1, r_2, \ldots r_n\}$
Output: Bit String s
Output $r_1[0 : \mathsf{LenPrefix}^{\beta}(\mathcal{R})]$

Figure 3.6. Subroutine algorithms for Meteor

once, meaning using a lower β is likely acceptable.

Because the model used in sampling is a generative one, the model maintains state on its previous inputs. Each distribution generated by the model is dependent on the values sampled from previous distributions. Additionally, the model requires an initial state to begin the generative process. This state is abstracted by the history parameter $\mathcal{H}$ passed to instances of Encode and Decode. This allows the distributions generated by each successful sampling of a covertext token c_i to remain synchronized between the two parties. We assume that the entire history $\mathcal{H}$ is maintained between the parties, including the initial state that primes the model.

The encoding algorithm loops through three stages until the entire message has been successfully encoded: (1) generating and applying the mask, (2) sampling a next output to append to the covertext, and (3) updating the state of the algorithm based on the output of the sampling event. In the first stage, the mask is computed as the output of a pseudorandom generator and is applied with the XOR operation. The resulting value, r is distributed uniformly in $[0, 2^{\beta+1})$, as each bit of r is distributed uniformly in $\{0, 1\}$. This random value is then used in step (2) to sample the next output of the sampling

Algorithm 3.4: $\mathsf{KeyGen}^{\beta}_{\mathcal{M}}$

Input: 1^{λ}
Output: Key k_{prg}
Output $k_{prg} \leftarrow \mathsf{PRG.Setup}(1^{\lambda})$

Algorithm 3.5: $\mathsf{Encode}^{\beta}_{\mathcal{M}}$

Input: Key k_{prg}, Plaintext Message m, History $\mathcal{H}$
Output: Stegotext Message c
$c \leftarrow \varepsilon, n \leftarrow 0$
while $n < |m|$ **do**
  $mask \leftarrow \mathsf{PRG.Next}(k_{prg})$
  $r \leftarrow m[n : n + \beta] \oplus mask$
  $c_i \leftarrow \mathsf{Sample}^{\beta}_{\mathcal{M}}(\mathcal{H}, r)$
  $\mathcal{R} \leftarrow \mathsf{Recover}^{\beta}_{\mathcal{M}}(\mathcal{H}, c_i)$
  $n_i \leftarrow \mathsf{LenPrefix}^{\beta}(\mathcal{R})$
  $c \leftarrow c \| c_i, n \leftarrow n + n_i, \ \mathcal{H} \leftarrow \mathcal{H} \| c_i$
Output c

Algorithm 3.6: $\mathsf{Decode}^{\beta}_{\mathcal{M}}$

Input: Key k_{prg}, Stegotext Message c, History $\mathcal{H}$
Output: Plaintext Message m
$x \leftarrow \varepsilon$
Parse c as $\{c_0, c_1, \ldots, c_{|c|-1}\}$
for $i \in \{0, 1, \ldots, |c| - 1\}$ **do**
  $\mathcal{R} \leftarrow \mathsf{Recover}^{\beta}_{\mathcal{M}}(\mathcal{H}, c_i)$
  $x_i \leftarrow \mathsf{Prefix}^{\beta}(\mathcal{R})$
  $mask \leftarrow \mathsf{PRG.Next}(k_{prg})$
  $x \leftarrow x \| (x_i \oplus mask[0 : |x_i|])$
  $\mathcal{H} \leftarrow \mathcal{H} \| c_i$
Output x

Figure 3.7. Symmetric steganography algorithms for Meteor

scheme. To determine the number of bits this sampling event has successfully encoded, the encoding algorithm uses the $\textsf{Recover}^\beta$ functionality of the RRRSS and calls $\textsf{LenPrefix}$ on the resulting (multi-)set. Finally, the algorithm then updates the β bits that will be used in the next iteration, and updates its other state as appropriate.

The decoding algorithm performs these same three stages, but with the order of the first two reversed. With knowledge of the output of each sampling stage c_i, the first algorithm calls $\textsf{Recover}^\beta$ and $\textsf{Prefix}$ to recompute some (possibly zero) leading bits of the r. Then, it calculates the mask that was used by the encoder for those bits and removes the mask. The bits recovered in this way make up the message.

Note that we do not discuss reseeding the $\textsf{PRG}$. Most $\textsf{PRG}$s have a maximum number of bits that can be extracted before they are no longer considered secure. Because the $\textsf{PRG}$ secret information is shared by the sender and receiver, they can perform a rekeying or key ratcheting function as necessary.

3.5.2.2 Correctness

Correctness follows directly from the properties of the RRRSS and the correctness of the PRG. We know that the RRRSS always will return the full set of random values that could have generated the sample, and thus recovery of the masked plaintext it deterministic. The receiver is able to recompute the same mask (and remove it) because of the correctness of the PRG, *i.e.*, it is also deterministic.

3.5.2.3 Security

We sketch the proof of security, as the formalities of this simple reduction are clear from the sketch. Consider an adversary $\mathcal{A}$ which has non-negligible advantage in the security game considered in Definition 1. We construct an adversary $\hat{\mathcal{A}}$ with non-negligible advantage in the $\textsf{PRG}$ real-or-random game, with oracle denoted $R(\cdot)$. To properly answer queries from $\mathcal{A}, \hat{\mathcal{A}}$ runs the encoding algorithm in Algorithm 2 with an

arbitrary input message, but queries the $R(\cdot)$ to obtain the mask required for sampling. Additionally, $\hat{\mathcal{A}}$ keeps a table of all queries sent by $\mathcal{A}$ and the responses. When $\mathcal{A}$ queries the decoding algorithm, $\hat{\mathcal{A}}$ checks its table to see if the query matches a previous encoding query, and responds only if it is an entry in the table. Note that if $R(\cdot)$ implements a true random function, the encoding algorithm simply samples a random message from the distribution. When $\mathcal{A}$ terminates, outputting a bit b, $\hat{\mathcal{A}}$ outputs b as well.

As the message is masked by the queries $\hat{\mathcal{A}}$ sends to $R(\cdot)$, $\mathcal{A}$ must be able to distinguish between a true-random output and the xor of a message with a one-time pad. Because XOR preserves the uniformly-random distribution of the pad, this is not possible with non-negligible probability.

3.5.2.4 Efficiency

We now show that the asymptotic expected throughput of Meteor is proportional to the entropy in the communication channel. Recall that the entropy in a distribution $\mathcal{P}$ is computed as $-\sum_{i\in|\mathcal{P}|} p_i \log_2(p_i)$, where p_i is the probability of the ith possible outcome of $\mathcal{P}$. Similarly, the expected throughput of Meteor can be computed as $\sum_{i\in|\mathcal{P}|} p_i \mathsf{Exp}(p_i)$, where $\mathsf{Exp}(\cdot)$ is the expected number of shared prefix bits for some continuous interval of size p_i. Thus, the remaining task is to compute a concrete bound on $\mathsf{Exp}(\cdot)$.

We will make the simplifying assumption that the start of an interval p_i is placed randomly between $[0, 2^{\beta+1})$. Note that interval i will never start after $2^{\beta+1} - p_i$ in practice, so we the number of prefix bits in this case to be 0, so this simplification will lead to an expected throughput strictly less than the true value. Additionally, the starting locations for each interval are not independent in practice, as they each depend on $p_{j\neq i}$. However, this independence assumption also leads to equal or lower expected throughput, as the starting point for larger intervals will actually be more biased towards the middle of the distribution, where $\mathsf{Exp}(\cdot)$ will be lower, and smaller distributions will be biased to start near the edges of the distribution, where $\mathsf{Exp}(\cdot)$ will be higher.

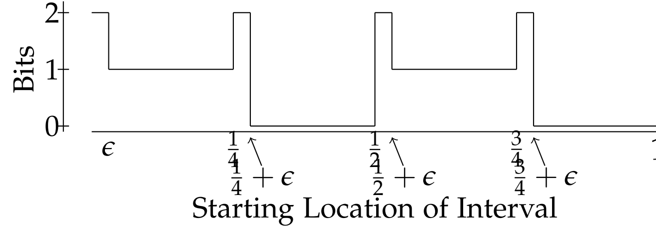


Figure 3.8. Bits of throughput by starting location for an interval i with size $p_i = \frac{1}{4} - \epsilon$, for some small ϵ. The expected throughput can be computed as the average of this function, *i.e.* $\mathsf{Exp}(p_i) \geq (2)(\frac{1}{4} - \epsilon)(0) + (2)(\frac{1}{4} - \epsilon)(1) + (2^2)(\epsilon)(2) = \frac{1}{2} - 6\epsilon$

By way of example, consider an interval i such that $p_i = \frac{1}{4} - \epsilon$, for some small ϵ (see Figure 3.8). If i starts between $[0, \epsilon)$, then it is contained completely before the prefix 01 begins, and thus would transmit 2 bits. The following p_i starting points all transmit only 1 bit, as the only shared prefix for the interval would be 0. If i starts between $[\frac{1}{4}2^{\beta+1}, (\frac{1}{4} + \epsilon)2^{\beta+1})$, the entire interval shares the prefix 01, so 2 bits can be transmitted. In $[(\frac{1}{4} + \epsilon)2^{\beta+1}, \frac{1}{2}2^{\beta+1})$, there is no shared prefix, as some of the samples that would land in that interval start with a 0 and others start with 1. The analysis continues in this way for the remainder of the starting points.

More generally, the expected throughput of an interval with size p is the average of these different sets of starting points with different length shared prefixed, weighted by size. More explicitly, let $g(p) = \lfloor -\log_2(p) \rfloor$, then

$$\mathsf{Exp}(p) \geq \begin{cases} 0 & , p > 1/2 \\ g(p)(2^{-g(p)} - p)2^{g(p)} + p\sum_{j=1}^{g(p)-1}(j2^j) & , p \leq 1/2 \end{cases}$$

The first part of the expression corresponds to the starting points where the interval has the most shared bits, *e.g.* the points in Figure 3.8 where the throughput is 2. There are $2^{g(p)}$ of these sets, each of which has size $(2^{-g(p)} - p)$, the difference between p and the nearest power of two less than 2. The sum corresponds to the when the interval transmits fewer bits, *e.g.* the points in Figure 3.8 where the throughput is 1 or 0. Each of these terms counts the $p2^j$ starting points where the number of bits transmitted is j.

Note that $\mathsf{Exp}(p) \geq \frac{1}{2}(-\log_2(p) - 1)$ for small enough p. To see this, note that

$g(p) \geq -\log_2(p) - 1$, because of the rounding. Then, just consider the first term

$$g(p)(2^{-g(p)} - p)2^{g(p)} \geq (-\log_2(p) - 1)(1 - p2^{-\log_2(p)-1})$$
$$= \frac{1}{2}(-\log_2(p) - 1).$$

While this bound is not tight, it illustrates that $\mathsf{Exp}(p)$ asymptotically acts like $\log_2(p)$, meaning $\sum_{i \in |\mathcal{P}|} p_i \mathsf{Exp}(p_i)$, grows proportionally to the entropy in $\mathcal{P}$, $-\sum_{i \in |\mathcal{P}|} p_i \log_2(p_i)$. Thus, the expected throughput of Meteor is asymptotically optimal.

3.6 Evaluation

In this section we discuss our implementation of Meteor and evaluate its efficiency using multiple models. We focus on evaluating Meteor, not a hybrid steganography system using the public key stegosystem in Section 3.4, because it is significantly more efficient. Moreover, the efficiency of a hybrid stegoanography system is determined by the efficiency of its constituent parts; the cost of such a scheme is simply the cost of transmitting a key with the public key scheme (see Section 3.4) plus the cost of transmitting the message with Meteor. An interactive online demonstration of our system is available at `https://meteorfrom.space`.

3.6.1 Implementation Details

We implemented Meteor using the PyTorch deep learning framework [159]. We realize the **PRG** functionality with `HMAC_DRBG`, a deterministic random bit generator defined in NIST SP 800-90 A Rev. 1 [17]. The implementation supports any type of binary data, such as UTF-8-encoded strings or image files, as input.

To illustrate Meteor's support for different model types, we implemented the algorithm with the weakened version of the GPT-2 language model released by OpenAI and two character-level recurrent neural networks (RNN) that we train. The GPT-2 model [168] is a generative model of the English language. It parses language into a vocabulary of

Algorithm 3.7: $\mathsf{Sample}^{\beta}_{\mathcal{M}}$ for the GPT-2 model.

Input: Randomness $r \in \{0,1\}^{\beta}$, History $\mathcal{H}$
Output: Token *next*
$\mathcal{T}, \mathcal{P} \leftarrow \mathsf{Next}_{\mathcal{M}}(\mathcal{H})$
$cuml \leftarrow 0$
for $i \in \{0, 1, \ldots, |\mathcal{T}| - 1\}$ **do**
    $cuml \leftarrow cuml + \mathcal{P}[i]$
    if $cuml > r$ **then**
        Output $next \leftarrow \mathcal{T}[i]$
Output $next \leftarrow \mathcal{T}[|\mathcal{T}| - 1]$

Algorithm 3.8: $\mathsf{Recover}^{\beta}_{\mathcal{M}}$ for the GPT-2 model.

Input: History $\mathcal{H}$, Sample s
Output: Randomness set $\mathcal{R}$
$\mathcal{T}, \mathcal{P} \leftarrow \mathsf{Next}_{\mathcal{M}}(\mathcal{H})$
$cuml \leftarrow 0$
for $i \in \{0, 1, \ldots, |\mathcal{T}| - 1\}$ **do**
    if $\mathcal{T}[i] = s$ **then**
        Output $\mathcal{R} \leftarrow \{r \in \{0,1\}^{\beta} \mid cuml \leq r < cuml + \mathcal{P}[i]\}$
    $cuml \leftarrow cuml + \mathcal{P}[i]$
Output $\mathcal{R} \leftarrow \emptyset$

Figure 3.9. RRRSS algorithms for the GPT-2 model. $\mathcal{T}$ is an array of possible next tokens and $\mathcal{P}$ is the probability associated with each of these tokens.

words and generates words when given previous context. Meteor encodes stegotext into these generated words. The character-level models generate ASCII characters in each iteration. These models output lower-quality English text, but are more generalizable. Character-level models work with any data that can be represented as text, including other languages and non-text protocols, whereas word-level models are specific to the English language models.

Our GPT-2 codebase builds upon that of [250]. We note that the next-generation GPT language model, GPT-3, has been published by OpenAI [33]; however, at the time of this writing, the codebase for the GPT-3 has not been released. The GPT-3 interface is the same as the GPT-2, meaning integration will be automatic, increasing stegotext quality

while maintaining security guarantees. Example stegotext generated with the GPT-2 model can be found in Section 3.6.4.

Figure 3.9 shows how to instantiate the $\mathsf{Sample}_{\mathcal{M}}^{\beta}$ and $\mathsf{Recover}_{\mathcal{M}}^{\beta}$ algorithms from Section 3.3 with the distribution represented as a generative model $\mathcal{M}$ (in discussion of classical steganography, we used $\mathcal{D}$). Both algorithms use $\mathsf{Next}_{\mathcal{M}}(\mathcal{H})$, which generates an array of possible next tokens $\mathcal{T}$ and an array of probabilities associated with each token $\mathcal{P}$ using the model's internal structure. The $\mathsf{Sample}_{\mathcal{M}}^{\beta}$ for generative networks accumulates the probabilities and selects the first token for which the cumulative probability exceeds the randomness supplied. This is equivalent to multinomial sampling, and is the unmodified method of sampling normally from the GPT-2 model. In the unmodified (i.e., non-Meteor) case, the GPT-2 chooses a true random value r instead of a PRG as in Meteor. $\mathsf{Recover}_{\mathcal{M}}^{\beta}$ inverts the process, returning the entire set of random values that would yield the target sample s.

In addition to the GPT-2 variant, we trained two character-level RNN models to test with Meteor, using the code of [173] with locally trained models. Each model uses long short term memory (LSTM) cells to store state [97]. The first model, named "Wikipedia", was trained on the Hutter Prize dataset [106], which consists of a subset of English Wikipedia articles. The data from this model contains English text structured with Wiki markup. The output of this model is good, but its character-level nature makes its outputs less convincing human text than GPT-2 output. The second model, named "HTTP Headers", consist of the headers for 530,128 HTTP GET requests from a 2014 ZMap scan of the internet IPv4 space [57, 66]. This highly structured dataset would facilitate hiding messages amongst other HTTP requests. We note that the flexibility of character-level models allows us to generalize both text-like channels and protocol-esque channels [112]. Both models have three hidden layers. The Wikipedia model has a hidden layer size 795 and was trained for 25,000 epochs. The HTTP headers model has size 512 and was for 5,000 epochs, due to its more structured nature. The two models were trained at a

batch size of 100 characters and learning rate 0.001. Example output from the Wikipedia character-level model can be found in Section 3.6.4.

3.6.2 Optimizations

In evaluating Meteor, we also implement two heuristic optimizations that could lead to better performance without compromising security. Note that while they increases the *expected* throughput of scheme, it is not guaranteed to do so. Making any change to the output selected in a given sampling event might unintentionally push the model down a lower entropy branch of the covertext space, yielding more sampling iterations overall. The first optimization is performing a deterministic reordering operation of the model distribution, reduces the number of calls to the generative model by 20%-25%, and in some cases results in more efficient encoding and decoding times. The second optimization is an adaptation from the NLP literature that uses the generative model's internal word representation to compress English language messages.

Before proceeding to the optimizations themselves, recall the intuition provided for Meteor in Section 3.5. In each iteration of the encoding algorithm, the sender extracts a probability distribution $\mathcal{P}$ from the generative model. $\mathcal{P}$ is subdivided into a series of continuous intervals $r_0, r_1, \ldots r_m$, the size of which determines the probability that the model would select the corresponding token is the next output. Meteor then generates a random sampling value $r = mask \oplus m$ and determines the interval r_i into which r falls. The number of bits encoded is computed as $\mathsf{LenPrefix}(r_i)$.

3.6.2.1 Optimization 1: Reordering the Distribution

We note that while we cannot manipulate $|r_i|$ without compromising the security of scheme, we are able to impact $\mathsf{LenPrefix}(r_i)$ by permuting the order of $r_0, r_1, \ldots, r_m$. It is clear there exists some such permutation that maximizes the expected throughput of Meteor, although finding this permutation proves to be difficult.

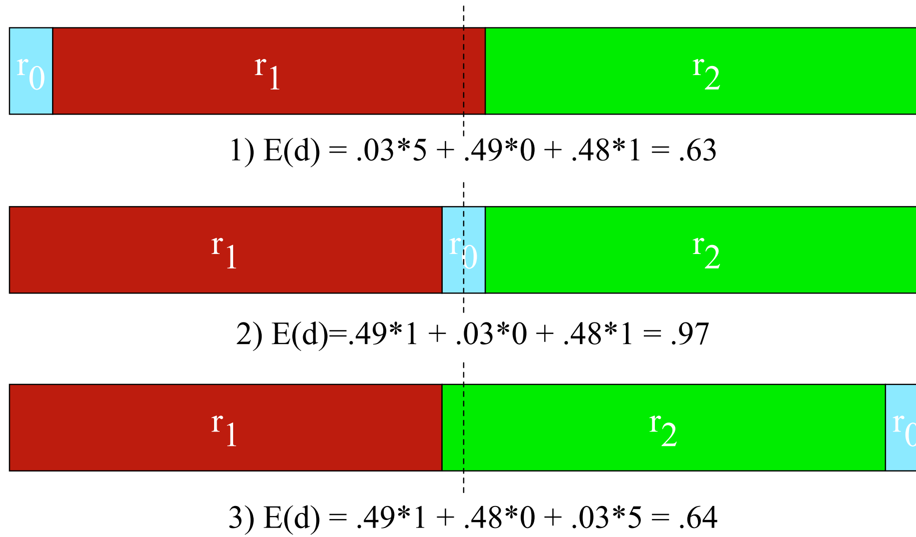


Figure 3.10. An illustrative example of the impact of reorganizing a distribution. r_0 has 3% of the total probability density, while r_1 and r_2 have 48% and 49% respectively. Because $2^{-6} < .03 < 2^{-5}$, r_0 can encode 5 bits of information when located at the beginning or end of the distribution. In orderings (a) and (c), one of the larger intervals crosses the 50% line, meaning $\mathsf{LenPrefix}(\cdot) = 0$. When the smallest interval is placed in the middle, the total expected throughput of the distribution rises.

The distribution $\mathcal{P}$ is generally output by the model in some sorted or lexicographic order. This might yield to some orderings of r_i that are incredibly unfavorable to $\mathsf{LenPrefix}(\cdot)$. Consider an illustrative example in Figure 3.10. If an interval r_i contains values on either side of the middle of the distribution, then $\mathsf{LenPrefix}(r_i) = 0$. When a large interval does so, as in cases (1) and (3), this severely decreases the expected number of bits that the distribution can encode. While this example is clearly contrived, it illustrates the impact correctly ordering $\mathcal{P}$ can have on the expected throughput – in this example an increase of over 50%. Importantly, we can use any reorganization procedure on the distribution provided (1) the same resulting permutation can be computed by both the sender and the receiver and (2) the size of r_i remains the same for all r_i.

Finding the optimal permutation of $\mathcal{P}$ proves to be a difficult task. Intuitively, each interval r_i, must be placed as a continuous block somewhere between 0 and 1 such that it does not overlap with other intervals. We take inspiration from approximation algorithms and design a greedy algorithm with pretty good performance, and we leave formal

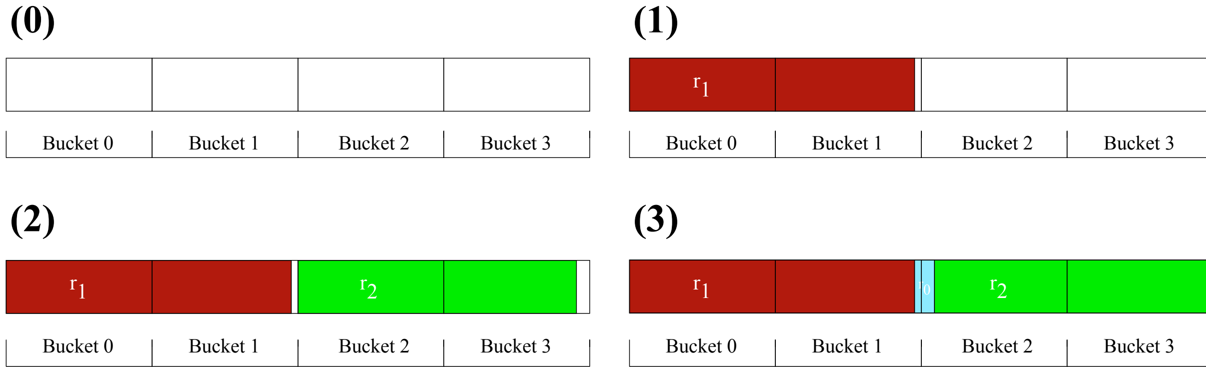


Figure 3.11. An overview of our reorganization algorithm. This distribution has entropy 1.16, so we create $2^2 = 4$ buckets. In (1), we place the largest interval r_1 into bucket 0, overflowing its value through most of bucket 1. Note that r_1 could have been placed in bucket 2; in general, we break ties by taking the earlier bucket. In (2), r_2 can be placed either in bucket 1, overflowing into the following buckets, or placed in bucket 2, overflowing into bucket 3. To maximize $\mathsf{LenPrefix}(r_2)$, we place it in bucket 2. Finally, in (3), we note that r_0 will not fit in bucket 3, so it must be placed in bucket 1. The pushes the later intervals, in this case r_2 down to make sufficient space.

analysis and bounds proving of this algorithm for future work. A simple algorithm would be to find a "starting point" to place each interval, starting with the largest, that maximizes $\mathsf{LenPrefix}(r_i)$. However, there are 2^β possible starting points, meaning a linear search will be prohibitively expensive. Instead we generate $2^{\lceil H(\mathcal{P}) \rceil}$ buckets with capacity $\frac{\sum_i (r_i)}{2^{\lceil H(\mathcal{P}) \rceil}}$, where $H(\mathcal{P})$ is the entropy in the distribution. These buckets represent potential "starting points" that each r_i can be placed. Note that the entropy represents an upper bound on the possible value of the expected throuhput $E(\mathcal{P})$ and if each interval r_i could perfectly fit into one of these bins, $E(\mathcal{P}) = H(\mathcal{P})$.

Starting with the largest r_i, we find the bin that will maximize $\mathsf{LenPrefix}(r_i)$ when r_i is appended to that bucket. As buckets become full, they are no longer options for placement. Note that r_i may exceed the remaining capacity of a bucket, or even the total capacity of a bucket. When this is the case, we "overflow" the remainder into the following buckets. Occasionally, this overflowing remainder may cause a chain reaction, requiring other, already placed intervals be "pushed" to make space. We give a simple example of our reorganization algorithm in Figure 3.11, using the same distribution given in Figure 3.10. Step (3) gives an example of overflow that causes one of these

chain reactions. Once each interval has been placed into a bin, the final ordering can be recovered by appending the contents of the bins.

The runtime of this algorithm is $O(2^{\lceil H(\mathcal{P}) \rceil} m)$, where m is the number of intervals; in our experiments, $\lceil H(\mathcal{P}) \rceil$ is typically less than 7, so this is close to $O(m)$, which is unsurprising given its similarities to bin-sorting. When n is very large, however, this algorithm is prohibitively expensive. In those cases, we use this algorithm to place the "big" intervals, and then simply place the smaller intervals into the first bucket with space.

3.6.2.2 Optimization 2: Compressing with Native Embedding

When encoding an English language message into a word-based, English model, we can use the model itself as a compression function. This optimization is implemented in the code of prior work, including [250]. All known words in the model's vocabulary are internally represented by a unique number. Before encoding, the secret message can be tokenized and each token can be replaced by its unique identifier. These identifiers are then parsed as bits and encoded as normal. This technique compresses the length of the message, thereby reducing the stegotext length required to send a message. However, this optimization is only useful if the underlying message is an English-like distribution; otherwise, the model cannot encode the plaintext in its internal representation.

3.6.3 Results

To measure performance across different hardware types, we evaluate Meteor on 3 systems: (1) *Desktop/GPU*, a Linux workstation with an Intel Core i7-6700 CPU, NVIDIA TITAN X GPU, and 8 GiB of RAM, (2) *Laptop/CPU*, a Linux laptop with an Intel Core i7-4700MQ CPU, no discrete GPU, and 8 GiB of RAM, and (3) *Mobile*, an iPhone X running iOS 13. The Desktop ran benchmarks on the GPU, while the Laptop machine ran on the CPU; as such, the Laptop is more representative of consumer hardware. We evaluate Meteor on both the Desktop and Laptop using each of the three models discussed above.

Additionally, we evaluate reordering and native compression optimizations (see below). The results are summarized in Table 3.2. We discuss mobile benchmarks separately at the end of this section.

3.6.3.1 Model Performance

The capacity, or number of bits encoded per token, is much higher for the GPT-2 model examples than for the Wikipedia and HTTP Headers models. Intuitively, the word-level nature of GPT-2 means there is usually more entropy in each distribution, whereas the character-level models have, at most, 100 printable ASCII characters from which to sample; this pushes the capacity of a single token to be much higher as a result. The stark difference in capacity between the capacities of Wikipedia and HTTP Headers can be attributed to the difference in structure of the training data. The Wikipedia dataset, although structured, is mostly English text. On the other hand, the HTTP Headers dataset is based on the HTTP protocol, which is rigid in structure — variation only exists in fields that can change, such as dates and URLs.

3.6.3.2 Encoding Statistics

Our next suite of benchmarks measures the relationship between the length of message and the time it takes to produce a stegotext. We generated plaintexts randomly and encoded them, incrementing the length of the message by one in each run. The results are plotted in Figure 3.12, which shows a clear linear relationship between the two variables. It is also apparent from the plot that the variance in encoding time increases as the length increases. This is because as tokens are selected, the model state can diverge; in some of these branches, the entropy may be very low, causing longer encoding times. This is amplified in the HTTP Headers model, as the baseline entropy is already very low.

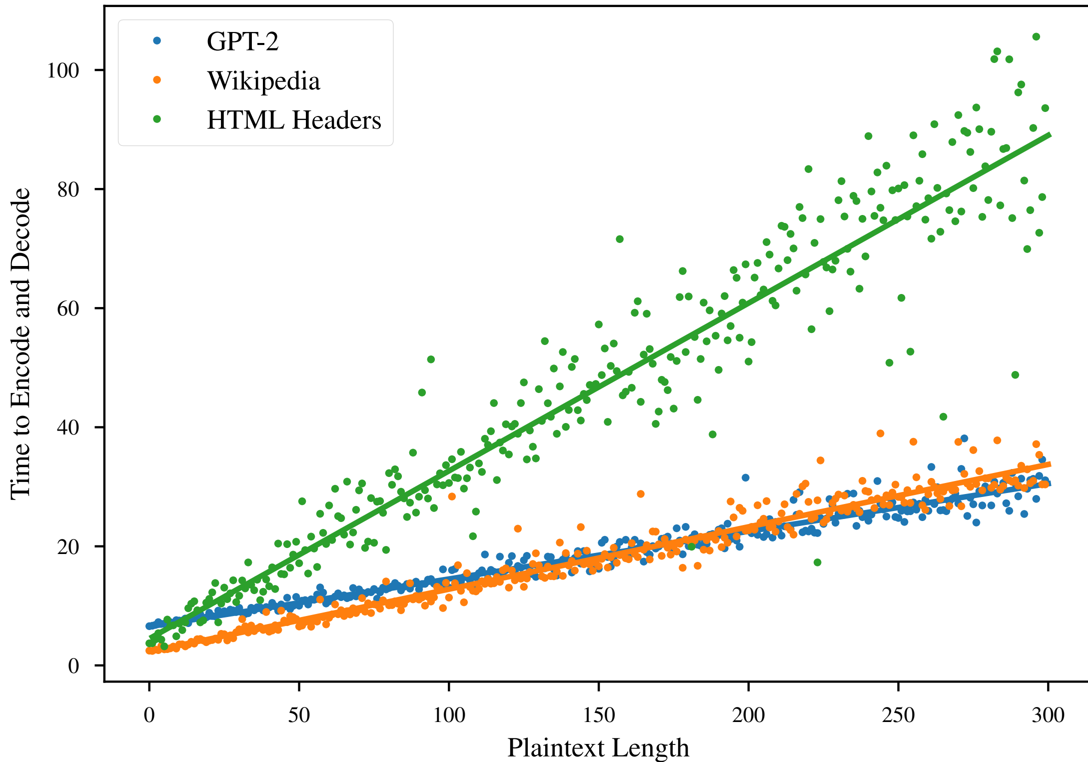


Figure 3.12. Comparison of plaintext length versus time to run encoding and decoding for different Meteor models. $R = 0.9745$ (GPT-2), 0.9709 (Wikipedia), 0.9502 (HTTP Headers)

3.6.3.3 Optimizations

In addition to implementing Meteor, we also evaluated the two heuristic optimizations discussed in Section 3.6.2 above that could yield shorter stegotext.

We evaluated the first optimization (Section 3.6.2.1) for all three of our models (see Table 3.2). For the GPT-2 model, we see a marked (24.8%) increase in capacity as well as a proportional reduction in stegotext length as a result of reordering the model outputs. The reordering does induce computational overhead, as the distribution over which the heuristic is performed is large (max 50,256 tokens). Reordering induces a 0.5% overhead in the Laptop/CPU, where updating the model is slow, and 69.0% overhead in the Desktop/GPU, where updating the model is fast. For the lower entropy models, the reordering algorithm we use is significantly faster, but yields mixed results. We believe these mixed results are an artifact of our choice of greedy reordering algorithms, which

Table 3.2. Model statistics for encoding a 160-byte plaintext. Timing results reflect model load, encoding, and decoding combined.

Mode	Desktop/GPU (sec)	Laptop/CPU (sec)	Stegotext Length (bytes)	Overhead (length)	Capacity (bits/token)
GPT-2	18.089	82.214	1976	12.36×	3.09
GPT-2 (Reorder)	30.570	82.638	1391	8.69×	4.11
GPT-2 (Compress)	11.070	42.942	938	3.39×	3.39
Wikipedia	19.791	46.583	2002	12.51×	0.64
Wikipedia (Reorder)	15.515	39.450	1547	9.67×	0.83
HTTP Headers	49.380	103.280	6144	38.4×	0.21
HTTP Headers (Reorder)	57.864	127.759	7237	45.23×	0.18

Table 3.3. Performance measurements for Meteor on the GPT-2 by device for a shorter context. Times are provided in seconds.

Device	Load	Encode	Decode	Overhead (time)
GPU	5.867	6.899	6.095	1×
CPU	5.234	41.221	40.334	4.6×
Mobile	1.830	473.58	457.57	49.5×

may perform poorly with heavily biased distributions.

We also evaluated the second optimization (Section 3.6.2.2). When implemented with GPT-2, we see a 47.77% decrease in time spent on CPU, and an associated 52.5% decrease in stegotext size. While powerful, this technique can only be used to encode English language messages into English language models. Compressing the plaintext message using traditional compression (*e.g.*, GZip) would yield similar results.

3.6.3.4 Mobile Benchmarks

Because Meteor is intended for censorship resistance, it is natural to benchmark it on mobile devices, where most sensitive communication happens. We implement Meteor on iOS using the CoreML framework, utilizing an existing GPT-2 iOS implementation as a base [104]. To our knowledge, our work represents the first evaluation of a neural network-based steganographic system on a mobile device. Our implementation, in Swift, employs an even smaller version of the GPT-2 model which fits on mobile devices as it uses smaller size context. An example of the output from this experiment can be found

Plaintext:
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aenean a lacus sed magna fermentum lobortis. Pellentesque et facilisis nibh. Donec sit amet odio metus.

Figure 3.13. The 160-byte plaintext used for the model outputs in this section.

in Section 3.6.4.

Our results are summarized in Table 3.3. The Mobile benchmark in the table was performed on the iPhone X Simulator, as we wished to instrument and profile our tests. We separately confirmed that simulator runtimes were similar to those of actual iPhone X hardware. While Laptop/CPU is 4.6× slower than Desktop/GPU, the Mobile runtime is a massive 49.5× slower than the baseline case. While deep learning is supported on mobile platforms like iOS, the intensive, iterative computations required by Meteor and other neural stegosystems are not performant on mobile systems. Nonetheless, our proof-of-concept demonstrates that Meteor could be used in a mobile context, and hardware improvements [195] would allow for secure communication between users even when available communication platforms do not offer end-to-end encryption, such as WeChat.

3.6.4 Sample Model Outputs

We now show sample stegotext outputs as generated by Meteor using several different model types. The plaintext associated with all of these outputs is the first 160 bytes of Lorem Ipsum (Figure 3.13). Figure 3.14 shows a truncated output for a stegotext generated using the Wikipedia model, which seems to have generated some kind of Wiki-markup contents page. Figures 3.15 and 3.16 are GPT-2 outputs for different contexts provided as input. Each output reads like a news article or book chapter. Representative output for the HTML headers model can be found in Figure 3.17. Finally, Figure 3.18 is a screenshot of Meteor running on the iPhone Simulator, generating stream-of-consciousness news text. Note that the context is shorter on the iPhone, as it can hold less state.

```
Haired the latter expand of the legal instance of the Imperial
 ↪  State of the American foal bridge, it is suspective that he
 ↪  was also notable to ensure that they produced a consolidate
 ↪  [[electricity]], the actual psychological cabinet [[Greece]]
 ↪  was the same time. It was born in many in the second [[tuak]]
 ↪  and [[timber]] at the idea of [[computer account|computer
 ↪  science]], which provides sales might therefore be in popular
 ↪  writings such as the European Investigation.

==Video games==
* [[Brown Town]]
* [[Sagan in the United States|Commonwealth of Western
 ↪  Technology]]
* [[Template International Society of Europe]]
* [[Commonwealth of Russia]]
* [[Commerce coverage in Sweden]]
* [[List of Constitution by Westman of the United Kingdom|British
 ↪  Politics]]
* [[List of California Court in 1981]]
* [[International Party of Diabetos Integrated Policy]]
* [[United Kingdom]]

===Overseas Roman Empire===
* [[Communist Council of Ireland]]
* [[Confucian States of the South Edition]]
* [[Conservative Party of the United Kingdom]]

[[Category:American states]]
[[Category:Conferences of India]]
```

Figure 3.14. Snippet of Meteor encoding of the first 160 bytes of Lorem Ipsum as generated by the Wikipedia model. The output was truncated to fit.

Context:
The Alvarez hypothesis posits that the mass extinction of the dinosaurs and many other living things during the Cretaceous-Paleogene extinction event was caused by the impact of a large asteroid on the Earth. Prior to 2013, it was commonly cited as having happened about 65 million years ago, but Renne and colleagues (2013) gave an updated value of 66 million years. Evidence indicates that the asteroid fell in the Yucatan Peninsula, at Chicxulub, Mexico. The hypothesis is named after the father-and-son team of scientists Luis and Walter Alvarez, who first suggested it in 1980. Shortly afterwards, and independently, the same was suggested by Dutch paleontologist Jan Smit.

Stegotext:
The picture in The Pale I HCR scientists' discussion now spans three dimensions. The first importance of the Yucatan Peninsula is demonstrated with the following conclusion: the Pliocene Earth has lost about seven times as much vegetation as the Jurassic in regular parts of the globe, from northern India to Siberia. Even since 1976, the continent has received or is already experiencing the worst mass extinction in recorded history since there were a million years last left from the extinction of Antarctic ice and the rapid spread of hydrocarbon-rich water as the Earth entered the Cretaceous Period. In turn, the world's oceans have been breathtakingly transformed and as a result the surrounding environments are especially vulnerable. The reason is obvious: there was no geological record of the presence of the Yucatan Peninsula in the Late Miocene epoch, which is the line that runs from the southern edge of South America to Siberia. The commander-in-chief of the Yucatan Peninsula, Frederick Nystrom, has determined that the area is to stay as barren as possible. As a result, individuals living on lake beds and on coastal areas have witnessed the loss of about 90 percent of their habitat. The Yucatan Peninsula consists of four zones, with two different habitats separated, each of which has experienced inseam damage. In one zone, along the northern shore of Lake Shemal, the retreating Tarahumara Ocean has been melted into a deep, seafloor called Nova Ravine, which south-east of the Yucatan Peninsula flows into Lake Isthmus, where there is an abundance of turtle life. A second beach, which lies at the far end of the peninsula, has been spewed down by a sea wall supporting Madagascar's Great Ocean Earthquake, 9,000 feet in magnitude and caused large numbers of deaths. The third zone, along the coast of Cancun and in Asuncion, is less severely affected. An estimated 16 percent of the continent is protected from the destruction of oceanic winds and floods. All three zones are in a state of catastrophic destruction. According to the definition provided by the National Commission on the extinction of the dinosaurs (Infection and Immunization in the Ind

Figure 3.15. The "Dinosaur" context and associated Meteor encoding of the first 160 bytes of Lorem Ipsum as generated by GPT-2. This is the context used in Figure 3.5.

Context:
Washington received his initial military training and command with the Virginia Regiment during the French and Indian War. He was later elected to the Virginia House of Burgesses and was named a delegate to the Continental Congress, where he was appointed Commanding General of the nation's Continental Army. Washington led American forces, allied with France, in the defeat of the British at Yorktown. Once victory for the United States was in hand in 1783, Washington resigned his commission.

Stegotext:
Washington was martyred in the battle of Cullman in 1788. Although down 19 men, yet his mission accomplished. He ended his life in 1896. When he died of disease, three of the men responsible said that Washington had been moved and was seen at the funeral placing him in his father's arms. Another said that Washington treated his wounds with "a study of a personal sense of injustice." (A 16th century German historian calls Washington a liar because of this website.) He was also a friend of Dr. Henry H. Jackson, who was notable as George Washington's physician and a collaborator, and recognizes him by name in the any books about him. The last surviving manuscript is from 1888 and contains a frank and truthful account of the Quakers' plight. One story states that while fighting in Whitesburg, Washington succumbed to pneumonia. He was 38 years old and according to a manuscript he got out the following year reports he grew old and fell in love. He also mentions a meeting with a woman who broke into his home and first went with him into a bath and gave him food and sleep. Three days later the woman left the room expecting him to eat her lunch and on that day he left home at 9:30 am in despair. He had not been to his bedside. On seeing this, he said a voice in him called out, "Your name is Jack. What is the girl?" Hamilton said the superior told him, "She was a layover in a bed and seven[Pg 209] feet below the bed where the general slept in very feminine attire. Nobody had time to look into her face. What was she to tell you about the general?"
A
Washington's Official Address to Congress with Americans May 17th, 1781
"I am the one to announce completely that I am a true Christian and an eloquent philosopher. I am not constrained

Figure 3.16. The "Washington" context and associated Meteor encoding of the first 160 bytes of Lorem Ipsum as generated by GPT-2. This is the encoding used throughout the benchmarks in Section 3.6.

```
HP/5.2.15^M
X-Pingback: http://dusppdames.com/xmlrpc.php^M
Link: ^M
^M
HTTP/1.1 302 Moved Temporarily^M
Date: Wed, 05 Nov 2014 11:57:22 GMT^M
Server: ^M
Content-length: 0^M
Connection: keep-alive^M
Keep-Alive: timeout=60, max=2000^M
Location: http://187.234.160.3/login.lp^M
Set-Cookie: xAuth_SESSION_ID=Cayaa6f3+fDejBq7NoO7rlAA=; path=/;
 ↪  ^M
Cache-control: no-cache="set-cookie"^M
^M
HTTP/1.1 401 Unauthorized^M
WWW-Authenticate: Basic realm="EchoLife Portal de Inicio"^M
Content-Type: text/html^M
Transfer-Encoding: chunked^M
Server: RomPager/4.07 UPnP/1.0^M
EXT:^M
^M
HTTP/1.1 400 Bad Request^M
Content-Type: text/html^M
Date: Wed, 05 Nov 2014 16:55:54 GMT^M
Connection: close^M
Content-Length: 39^M
^M
HTTP/1.1 200 OK^M
Date: Wed, 05 Nov 2014 16:16:49 GMT^M
Server: Apache^M
Accept-Ranges: bytes^M
Connection: Keep-Alive^M
Keep-Alive: timeout=60, max=2000^M
Location: http://189.136.200.30/login.lp^M
Set-Cookie: xAuth_SESSION_ID=7rnefC8G+AxYersVCwwA=; path=/; ^M
Cache-control: no-cache="set-cookie"^M
^M
```

Figure 3.17. Snippet of Meteor encoding of Figure 3.13 as generated by the HTTP Headers model. The output was truncated to fit.

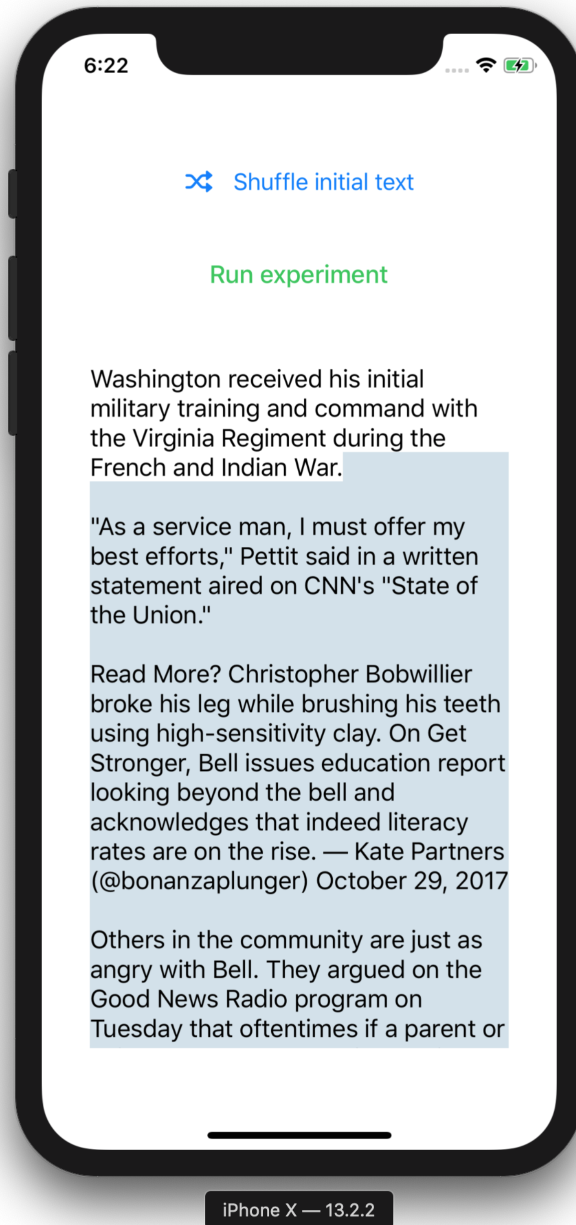


Figure 3.18. iPhone X screenshot of Meteor encoding of the first 160 bytes of Lorem Ipsum as generated by the GPT-2 model. Generated text is highlighted, and context is unhighlighted.

3.7 Comparison to NLP-Based Steganography

Noting the appeal of hiding sensitive messages in natural text, researchers in the field of natural language processing (NLP) have recently initiated an independent study of steganography. Unfortunately, this work does not carefully address the security implications of developing steganographic systems from NLP models. Instead, the results employ a variety of ad-hoc techniques for embedding secret messages into the output of sophisticated models. The resulting papers, often published in top NLP conferences, lack rigorous security analyses; indeed, existing work cannot be proven secure under the definitions common in the cryptographic literature. Highlighting this weakness, there is a concurrent line of work in the same conferences showing concrete attacks on these schemes, *e.g.*, [121, 139, 224, 235, 237, 238].

The first wave of steganographic techniques in the NLP community leverages synonyms and grammatical reorganization for encoding, *e.g.*, [42, 43, 92, 103, 192, 243]. The key observation in this work is that natural variation in linguistic patterns can be used to hide information. For instance, if one of two synonyms can be used in a sentence, each with probability .5, then the selection conveys a bit of information. Similarly, comma usage or word order can be used to encode small amounts of information. Because not all possible linguistic variations occur with equal likelihood, some of these works adapt a Huffman encoding scheme to facilitate variable length encoding, *e.g.*, [43, 92]. These approaches rely on linguistic idiosyncrasies and are therefore not generalizable.

More recently, researchers found ways to use the structure of these models to steganographically encode information, including LSTMs [75], Generative Adversarial Networks [214], Markov Models [236], and other forms of Deep Neural Networks [59, 230, 234, 250]. Rather than give an exhaustive description of the encoding techniques used in these works, we give a brief description of the most important techniques.

Early constructions directly modified the distributions. One such construction [75]

organized the distribution into "bins," each representing a short bitstring, and randomly selected an output from the bins corresponding to the message.[2] Building on this intuition, other research [59, 234] uses Huffman coding to encode variable numbers of bit in each iteration. More recent work has attempted to use the message itself as the sampling method, a method known as "arithmetic coding" [250]. This method attempts to convert a plaintext message into a deterministic stegotext based on its contents, iteratively using bits from the message to sample into the distribution. The first two constructions heavily modify the output distribution, rendering stegotext easily detectable. The arithmetic construction is also insecure, since it reuses randomness in multiple sampling events, a problem similar to the one that Meteor is designed to overcome.

The relaxed adversarial models considered in the NLP community lead to significantly less robust constructions. For instance, the adversaries in the NLP literature do not have access to the model [59, 75, 234, 250], significantly limiting the attacks they can mount. Without this assumption, an adversary can clearly differentiate between a stegotext and covertext by identifying biases in the output distribution. The adversary compares the candidate output to random samples from the model, easily distinguishing when a stegosystem is being run and defeating the purpose entirely.

The NLP threat model folds in the face of an advanced, persistent adversary who can always exfiltrate the model through other means. Moreover, recent advanced in adversarial machine learning have demonstrated how even the "secret" parameters of a black-box model can be extracted by seeing enough output [111, 155, 181], unlike that of encryption keys or pseudorandom functions. This pervasive requirement that the model remains private information is therefore unreasonable. Unable to achieve cryptographic security, these constructions evaluate their work by measuring the statistical difference between the output produced by the encoding scheme and real text. Highlighting the weaknesses of these schemes, numerous attack papers have been published, *e.g.*,

[2]A similar, but secure, partition based approach is investigated in [35]

Table 3.4. Comparative distribution statistics for samples from neural steganography algorithms in prior NLP work, with random sampling as a baseline. "N/A" indicates that a metric is not relevant for an algorithm.

Algorithm	Perplexity	KL-Divergence	Capacity	Entropy	Secure?
Meteor (this)	21.60	0.045	3.09	6.30	✓
Arithmetic [250]	29.22	0.082	4.82	6.66	✗
Huffman [59, 234]	8.85	0.851	2.31	N/A	✗
Bins [75]	50.82	2.594	3.00	N/A	✗
Random Sample	13.82	0.040	N/A	5.36	N/A

[121, 139, 224, 235, 237, 238]. These attacks use machine learning techniques to detect the presence of encoded messages generated with some of the works listed previously. Ad-hoc and non-cryptographic security is insufficient to provide security against powerful and determined adversaries, especially nation-state adversaries.

3.7.1 Comparative Analysis

We assess Meteor against the following previous solutions: (1) bins [75], (2) Huffman coding [234], and (3) arithmetic coding [250]. We compare standard NLP language statistics for these with a regular, random sample from the model, and provide our results in Table 3.4. Note that we mark entropy as "N/A" for Huffman and bins because these methods use a binning algorithm which prevents us from calculating entropy meaningfully. The random sample is a control distribution, and is not encoding anything thereby having "N/A" capacity.

Of particular note in our results is the Kullback-Leibler (KL) divergence across algorithms, which in this case compares the distribution of the model to the output distribution of the algorithm. The KL-divergence for Meteor is very close to that of the random sample, as Meteor merely changes the randomness to steganographically-encoded randomness. As discussed previously, algorithms that modify distributions from the model have high biases, and this is reflected in the KL-divergence of Huffman and bins being much higher than the rest. The arithmetic algorithm has a lower KL-divergence than the rest of the

NLP algorithms, as it does not modify the distribution; however, it has a higher value than Meteor because it reuses randomness, while Meteor uses fresh randomness like the baseline random sample does.

We also note that the security properties of Meteor do not hamper the capacity metric significantly. Arithmetic output has a higher capacity, but we note that the insecurity of this system makes this additional capacity moot; modifying the parameters to Huffman or bins could have yielded the same capacity with the same security vulnerabilities. Table 3.4 also includes perplexity and entropy statistics, that show Meteor is competitive in performance with the insecure primitives proposed previously.

Chapter 4

Building At-Home Cryptography

In this chapter, we consider at-home cryptography, and how it can be built to provide authentication and encryption services via a user's smart home devices.

4.1 Introduction

To mitigate the risk posed by device loss, users should be able to *voluntarily* restrict access to critical key material to times when their device is in some trustworthy *location*, *e.g.*, the home. These users can *opt-in* to restricting their usage, possibly because they may consider certain actions too sensitive to operate outside of a secure location or might consider themselves particularly at-risk. For instance, users might already limit their use of online banking or telemedicine to times when they are at home for privacy reasons. By limiting their use in this way, these users actively engage in misuse resistance, protecting themselves from risks outside of the home.

Users may also wish to utilize recently proposed privacy enhancing systems that assume the existence of a personal, fixed storage for secrets local to a user. For example, BurnBox [210] provides *self-revocable* encryption, which allows users to temporarily delete keys that could decrypt sensitive cloud data (*e.g.*, before a border crossing), and recovers these keys after the user is safely home with key material stored there.

The fundamental building block required to realize these applications is computation

that can only be performed at home, which can then be leveraged to perform cryptographic operations. We refer to such a system as one that provides *at-home cryptography*.

Importantly, mobile devices cannot facilitate at-home cryptography, as they cannot offer fine-grained access control mechanisms based on location; even if the user only uses the material while at home, that material is still on device—and therefore exposed—while they are on the go. Similarly, mobile devices cannot be the fixed store of secrets required for advanced privacy systems.

4.1.1 Prior Attempts

At first glance, achieving limited access to keys might appear trivial; a user can simply store key material for authentication or encryption on a device that *stays* at home rather than on their mobile device, *e.g.*, a desktop computer or a dedicated hardware security module. Because this device is stationary and only accessible on a local network—or even air-gapped—access to the key material is inherently limited.

While straightforward, this solution requires (1) the user to own stationary hardware, and (2) the user must have the technical expertise to manage their stationary hardware. In the time before widespread smartphone use, this solution made sense; personal computers were not very portable, and required at least some level of technical expertise to operate. However, this ostensibly simple solution is becoming unworkable for a rising part of the general population. 15% of adults in the United States only use smartphones as their primary device, with an upward trend since 2013; in the youngest generation of adults, 18-29 years old, this proportion increases to 28% [164].

Past research efforts also focus on the use of dedicated hardware to build this functionality. While *location-based cryptography* (of which at-home cryptography is a subset) has been studied in both the theoretical [32, 34, 41, 113, 160, 165] and applied [6, 77, 166, 186, 199] literature, no practical constructions have been realized or

widely deployed.[1] As such, we turn to a more pragmatic approach: *re-use* the devices users already have.

4.1.1.1 Re-using Devices for At-Home Cryptography

Although users are unlikely to have access to dedicated hardware for at-home cryptography, users may have access to Internet-of-Things (IoT) devices. These devices typically do not leave the home, making them an attractive prospect for anchoring a trusted computing base for at-home cryptography.

Re-using these devices raises its own difficulties, however:

- IoT devices are designed to be *single-purpose*, and, to keep costs low, have just enough compute capability to provide their application, unlike the general purpose capabilities of smartphones and computers. Re-using an IoT device beyond its intended purpose may induce measurable overhead, so we must ensure that at-home cryptography operations are lightweight.
- IoT devices have a *history of vulnerabilities* [62, 188], so it is not advisable to use one as a single store of secrets. Instead, we observe that it is more appropriate to distribute trust among many IoT devices, such that an attacker would need to compromise many IoT devices before exposing any of the user's secret data. Of course, resource constraints limit the viable approaches to federating trust.

As such, leveraging IoT devices into a practical at-home cryptography system requires carefully navigating tradeoffs between functionality, deployability, and security.

4.1.2 SocIoTy

In this work we present *SocIoTy*, a system design and protocol for at-home cryptography using a user's existing IoT devices. SocIoTy is designed for non-expert users who want to

[1] Indeed, there are impossibility results that might rule out "ideal" solutions [41].

protect high-value digital resources but do not have the access, expertise, or inclination to use dedicated hardware. Our system allows these users to set their own risk tolerances, allowing them to tie whatever secrets they consider to be most valuable to their smart home. While we focus on the smart home in this work, SocIoTy has applications wherever there are multiple embedded devices running on the same network, such as small businesses and hospitals. Our design is summarized in Figure 4.1.

SocIoTy builds an at-home cryptographic system for a pseudorandom function (PRF) [86], a simple, but powerful, primitive. From this at-home PRF, we can directly build two-factor authentication [147, 148] and derive keys for encryption. SocIoTy treats the smart home as a PRF that users can query to provide at-home cryptographic services. Because the user is physically at home, they can generate PRF outputs, and use these outputs to address their real-world needs—like generation of 2FA OTPs for authentication and of keys for cloud-encrypted content—all without worrying that their credentials are at risk outside of the home. Moreover, the interface to users is the same, and service providers would not have to change their architectures to accommodate SocIoTy; users perform one setup step on their smart home, and service providers only need to use a different PRF library in their backends.

To address the problems of IoT devices discussed above, we build a threshold, distributed PRF [149]: each IoT device computes a partial evaluation on an input, and a more powerful device (*e.g.*, a smartphone) reconstructs the actual PRF result from multiple partial evaluations and its own key material. This federates trust amongst all of the devices in the smart home, while reducing the computation power required from individual devices to one operation (in implementation, a single elliptic curve multiplication) for each user request.

We evaluate SocIoTy on a simulated smart home, consisting of analogs of smart home devices, from high-end, full-size systems (Raspberry Pis) to tiny, embedded microcontrollers (ESP32s). We collect microbenchmarks on these devices, as well as benchmarks

on full deployments of the system in realistic configurations. To highlight the ease of use of our proposed system, we also build a simple Google Authenticator-style smartphone app that uses SocIoTy to calculate OTPs. We find that our implementation meets the performance needs of our envisioned applications, while remaining seamless to the end user—performing OTP generation, for example, in < 200 milliseconds on average when involving a smartphone and 9 SocIoTy devices.

4.1.3 Contributions

In this work, we study the problem of giving non-expert users location-based access control to their cryptographic material, focusing on the smart home setting. Our goal is to help average users mitigate the risk associated with carrying high-value cryptographic material on their mobile devices, giving users the peace-of-mind in knowing that their service or files can only be accessed from home. Specifically,

1. We discuss *at-home cryptography*, highlighting relevant use cases and design considerations for an at-home cryptographic system (Section 4.3).
2. We present *SocIoTy*, an at-home cryptography system designed for non-expert users and their smart homes, and show how it can be used to build relevant constructions such as time-based one-time passwords [148] and self-revocable encryption [210] that are tied to the home (Section 4.4).
3. We implement and evaluate SocIoTy on realistic hardware, providing microbenchmarks for individual cryptographic operations on representative IoT devices and full benchmarks of an end-to-end deployment on a realistic smart home configuration (Section 4.5).

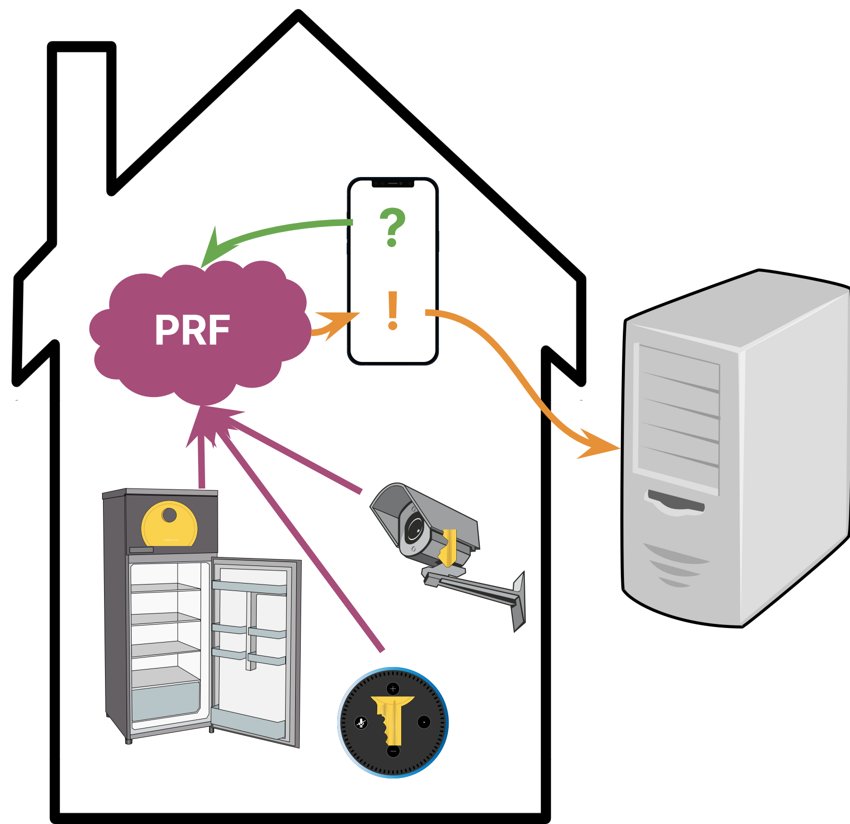


Figure 4.1. An overview of our SocIoTy, which uses a PRF built from IoT devices to provide at-home cryptographic services.

4.2 Background

4.2.1 Smart Homes

SocIoTy relies on a home Internet-of-Things network, or "smart home", to bootstrap at-home cryptography. IoT devices and smart home networks are proliferating rapidly [146]. In 2022, 57.5 million Americans lived in a smart home accounting for 45% of US households, and it is estimated that by 2026, more than 25% of homes worldwide will have some degree of IoT capability [200]. IoT devices range in computational capacity from extremely lightweight micro-controllers to fully-Linux-capable system boards with gigabytes of RAM. We rely on IoT devices to perform cryptographic operations and to communicate over the network in order to manifest a cryptographic scheme from the participation of otherwise logically isolated systems. As in prior work (*e.g.*, [125]), we assume a network of constrained devices (in terms of computation and power) participates in the cryptographic protocol, and in our evaluation (Section 4.5) model such devices to demonstrate feasibility.

4.2.2 Pseudorandom Functions (PRFs)

A pseudorandom function [86] is one which outputs values that, without knowing some secret sk, cannot be distinguished from random. PRFs can be used to build many other primitives in cryptography, including symmetric encryption and authentication schemes. For encryption, there exists a well-known theoretical construction that randomly chooses input for the PRF and treat the output as a one-time pad for the message. It is also possible to treat the output of a PRF as input to a key derivation function for block ciphers. Authentication is straightforward, as the input to the PRF can be the message the party would like to have verified, and the output being the tag for verification.

4.2.2.1 (Threshold) Distributed PRFs

One useful variant of a pseudorandom function is the distributed pseudorandom function (DPRF), which allow a group to *jointly* evaluate a PRF. Each party uses shares of the secret key to calculate a partial output that can later be combined to recover the full PRF output. This can be extended to the threshold case, where the computation is successful if t parties supply honest recovery values, but to any group of $t-1$ parties the PRF output appears to be uniformly random, creating a threshold DPRF (TDPRF). While TDPRFs can be constructed using generic MPC, it would be highly inefficient and take multiple rounds of communication to produce a result, both non-starters for IoT devices. We instead utilize a protocol that requires only one round of communication between evaluators and an aggregator, with no communication required between evaluators [149]. The protocol is based on the decisional Diffie-Hellman assumption and the use of random oracles.

The interface of a TDPRF consists of a tuple of algorithms (Gen, $\mathsf{PartialEval}$, Recon):

- $\mathsf{Gen}(1^\lambda, K, t, n)$ produces shares of the PRF key K denoted as $K_1 \ldots K_n$.
- $\mathsf{PartialEval}(K_i, x)$ uses a key share K_i on an input x to produce a partial evaluation of the PRF, y_i.
- $\mathsf{Recon}(\{y_i\}_{i \in Y})$ takes a subset of partial evaluations by users $Y \subseteq [n]$ where $|Y| \geq t$ and produces the full PRF output y.

Finally, we note that a TDPRF may have an additional efficient algorithm which takes in a fully reconstructed key K and an input x which we denote by $\mathsf{Eval}(K, x)$, allowing a TDPRF to be used as a regular PRF.

4.2.3 Two-Factor Authentication (2FA)

To increase user security, online service providers have started to roll out 2FA, which requires a second form of authentication to log in to a service. The most common form of 2FA after email and SMS [49] is the time-based one-time password (time-based OTP or TOTP) [148]. Every tc seconds, a user's token (*e.g.*, a smartphone app) generates an OTP. When the user wishes to authenticate, they input their username, password, and OTP. Unlike passwords, OTPs are short lived; they are only valid for the time interval tc in which they are generated, and can only be used once. Users therefore authenticate with either *something they know* (a password) or *something they are* (a fingerprint or retina scan), alongside *something they have* (their token, which generates OTPs). TOTP is supported by major social media platforms [220], electronic health records systems [65], financial institutions [174], and corporations [51]

The security of TOTP relies on the underlying HMAC-based OTP algorithm (HOTP) [147], which generates OTPs using the HMAC construction [124]. The security of HOTP, in turn, relies on the assumption that HMAC is a PRF. Since adversaries without sk cannot predict PRF outputs, they also cannot predict OTPs. Thus, as long as we assume that our underlying primitive (HMAC) is a PRF, then the OTPs generated by HOTP (and TOTP) are secure.

More formally, a TOTP is parameterized by tc and defined by $\mathsf{TOTP}_{tc}(sk, ts) = \mathsf{PRF}(sk, \lfloor \frac{ts}{tc} \rfloor) \mod 10^6$, where ts is a timestamp and $\mod$ is the modulus operation (used to convert the output of PRF into a 6-digit integer). We omit tc in our notation for TOTP for simplicity, and use the recommended default $tc = 30$.

4.2.3.1 Compelled Access

Compelled access to a software system or to data, whether by a malicious attacker or law enforcement agent, poses a serious risk to privacy and security. Compelled access can be

viewed as exploiting a user's ability to authenticate, and similarly compelled decryption can be viewed as exploiting a user's ability to decrypt sensitive data. Recent work has explored the mechanisms and mitigations of compelled access and decryption in mobile devices [251], as well as defending cryptographic protocols from compelled decryption by identifying and reducing long-lived secret values [185]. BurnBox [210] attempts to address compelled decryption by putting a user's ability to decrypt their files in escrow in a secure location—specifically, by allowing for a form secure deletion (*recovation*) which is reversible only with a secret key saved elsewhere, *e.g.*, in a vault at home.

Location-based cryptography is a powerful mechanism when considering compelled access and decryption. Broadly, location-based cryptography ties cryptographic operations to some notion of location [41, 165, 186]. If cryptography is only possible within a secure location (*e.g.*, the home), and compelled access or decryption can be expected to occur elsewhere (*e.g.*, at a border crossing or the proverbial dark alley), these risks are mitigated. Better yet, the user cannot be directly coerced (*i.e.*, via "rubber-hose" cryptanalysis) to release a key which is only accessible from a given remote location.

4.3 Designing At-Home Cryptography

To capture the notion that some cryptographic operations should only be available at home, *i.e.*, at-home cryptography, it is necessary to modify the interface to cryptographic calls with a location input. This modification clearly captures generic location-based cryptography, a superset of at-home cryptography. As we are only interested in this subset, we (informally) modify a cryptographic function F with input z to produce the function F_{home} as follows:

$$F_{\text{home}}(z, \mathsf{loc}) = \begin{cases} F(z) & \text{if } \mathsf{loc} = \text{home} \\ \perp & \text{otherwise} \end{cases}$$

We emphasize that this notation is informal; by making the location an input to the function, an adversarial caller could call the function with a location other than their

own. Formally modeling this transformation would require limiting the caller to use their true location, perhaps by letting users make queries to a subset of functionalities, where the subset is determined by their present location. Indeed, this better matches our envisioned system, in which these oracles are realized by distributed computation on hardware segmented to only a local network. In either sense, providing a formal framework for at-home cryptography is beyond the scope of this work; we will use the informal notation described above, as the intuitive meaning is clear.

4.3.1 Case Studies

To make the envisioned usage of at-home cryptography clear, we briefly present several concrete use cases. While not true anecdotes, these motivating examples are rooted in real-world trends and contextualize the technical considerations that must go into designing our at-home cryptography solution, SocIoTy.

4.3.1.1 Use Case 1: The Remote Worker

Consider a user that recently accepted a job offer from a prominent law firm as a legal aid, where they will work as a remote-only employee; this kind of remote-only work has been on the rise since the COVID-19 pandemic [158], and some anticipate that many of these jobs will remain fully-remote permanently [175]. To access the sensitive legal documents required to do their job, the user connects to the law firm's network over a VPN. To authenticate to the VPN, the user enters codes generated by a 2FA app on their company-managed smartphone. Company policy requires that the user should only connect to the VPN when within their home, owing to the sensitive nature of the company's documents. However, the user has no way of ensuring that they meet that policy if their smartphone is lost or stolen.

4.3.1.2 Use Case 2: The Outpatient

Consider a user who has end-stage renal disease, and requires active management through dialysis. Instead of remaining in the hospital, the user owns a dialysis machine at home, which are increasingly common [133]. The user regularly meets with their doctor to discuss their condition. On days they are not able to visit their doctor for a check-up, they set up a telemedicine appointment from home. Based on the check-up, the user's doctor is able to remotely configure the dialysis machine over the Internet. The user has an app they use to connect to hospital's electronic medical records system, but is nervous about their health data leaking when they leave the house.

4.3.1.3 Use Case 3: The Foreign Correspondent

Consider a user who is an investigative journalist that frequently travels to war-torn, authoritarian countries as part of their reporting duties. During such trips, the user keeps detailed notes, initial research, article drafts, and the identities of sources on their smartphone. To ensure that this information is not lost if their phone is lost, they back up these files to cloud storage services; due to the sensitive nature of these documents, they keeps them encrypted while in cloud storage and keeps decryption keys on their smartphone. While traveling, the user is often stopped and searched by local law enforcement (either at border crossings or during routine encounters on the street); such stops are common in countries with repressive regimes, and border officers are known to extract data from smartphones at border crossings [251]. The user has heard of next-generation cryptographic systems designed to let them temporarily revoke access to their sensitive documents until they return to a secure location (*e.g.*, [210]), but they lack the dedicated, stationary hardware those systems require.

4.3.2 Design Goals

With these use cases in mind, we now discuss the goals and considerations for a realization of an at-home cryptography system.

4.3.2.1 Functionality

All of the use cases in Section 4.3.1 require limiting execution of cryptographic functions to the home. One way of implementing this constraint is to only hold the secrets at home, so the required key material is unavailable in any other location. This would allow all three of our envisioned users to opt-in to limiting access; each envisioned user either does not require access on-the-go, or would like to ensure it is not possible.

We require support for both authentication (use cases 1 and 2) and encryption (use case 3). An authentication primitive means that we can use the home as a second factor for 2FA—*somewhere you are* in addition to the typical *something you know, have,* or *are* triad. An encryption primitive would allows users to secure files such that they can only be decrypted when the user is at home, suitable for privacy systems that require a digital safe [210] to recover files after a threat has passed and they have returned home. Note that it is possible to accomplish this task robustly while still storing encrypted files in the cloud—even if the encrypted files are available globally, the plaintext documents are location-bound.

4.3.2.2 Deployability

Prior work on location-based cryptography [41, 165, 186, 199] has not been deployed in practice due to its reliance on specific hardware to provide security properties or non-standard adversarial models. Therefore, we aim to use *existing* devices to achieve the location-binding property: namely, the IoT devices of the user's smart home. Since we are re-using devices, we must ensure that our solution does not require intensive or long-running computations on the IoT devices. They should achieve their cryptographic

task quickly and return to their primary functionality in the home. From the user's perspective, the only change is that the location matters for the task at hand; the rest of the interface for cryptography should be the same, and be fast enough that the user does not notice any latency.

We note that this does not preclude a more powerful device from being involved. The IoT devices can operate a lightweight part of the computation; then, *another* device—a smartphone or tablet—performs the more heavyweight computation. This other device and its interface can be the same as what would be used in a more traditional, non-location-bound cryptographic solution (*e.g.*, a 2FA app on a smartphone), abstracting away the new at-home cryptography system.

4.3.2.3 Security

An at-home cryptography system must be secure in the context of adversaries that are able to corrupt and control the smart home's devices, and those that are able to compel access to the user's secrets outside of the home. We more concretely define our threat model in Section 4.3.3.

The smart home setting introduces particular challenges not captured by traditional models. For example, family members and roommates can also share the space, perhaps with their own IoT devices. Additionally, each member of the household may have several different at-home services they wish to use. Any solution must be therefore secure in the presence of several *other* users and multiple *different* services.

We emphasise that our intention is to allow for users to *opt-in* to this extra layer of protection for the selected services that make sense for them (or the organizations of which they are a part); critically, these choices are highly contextualized to each user. In use case 2, for instance, the user's fixed medical devices should *only* be able to communicate at home; any communication outside the home would likely be an error. Moreover, the choice to location-bind access to particular services might also change over

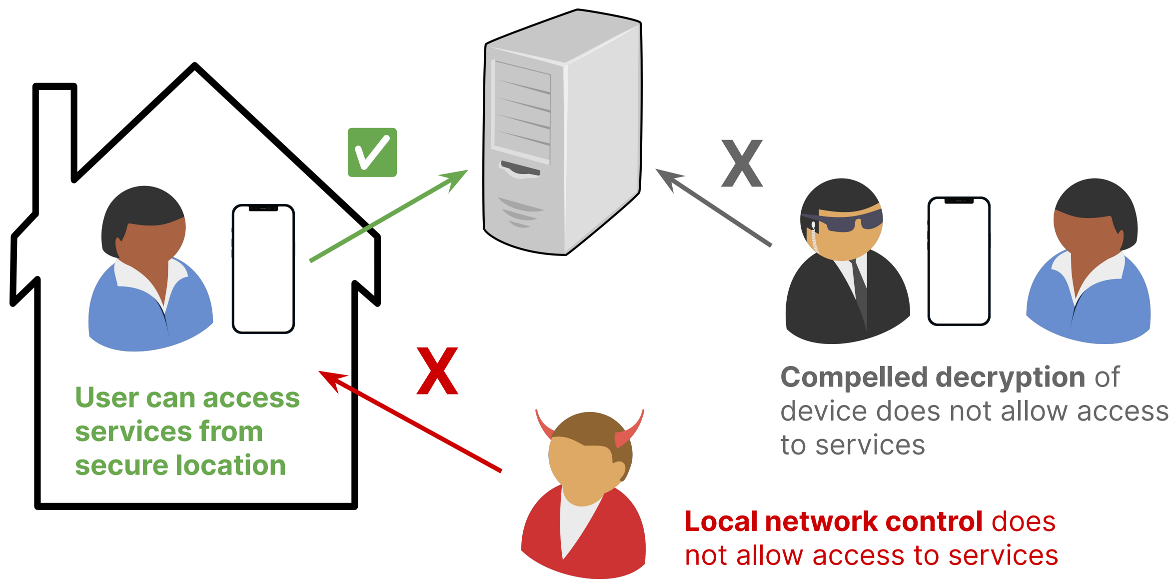


Figure 4.2. An overview of our threat model for SocIoTy.

time, based on what the user plans to do when they leave their home and the specific threats that they might expect along their journey. For example, in use case 3, the user might want to add an additional layer of security to their sensitive services only when planning to cross international borders, even if they do not location-bind access to these services in their daily life.

4.3.3 Threat Model

Since our system makes use of multiple devices and a variety of scenarios, it is important that our threat model systematically considers all of these components. We model around a setting where the user has IoT devices in their smart home, as well as a powerful mobile device (a smartphone or tablet) that can communicate with the IoT devices and can be involved in a setup procedure that authorizes it to participate in the protocol. We will refer to this device as the *authorized device*. To successfully tie cryptographic services to the home, the system must be designed in such a way that a cryptographic operation cannot succeed if this authorized device is not also at home and participating in the

protocol.

Concretely, we demonstrate this by showing that the protocol must be secure against the following types of adversaries:

4.3.3.1 Compelled Access Adversaries

We consider adversaries that can obtain access to the authorized device when it is not physically present in the home [251]. These adversaries can extract all of the secrets from the authorized device. For example, consider a border control officer that compels decryption of the user's authorized device while the user is crossing the border [210]. While they now have access to any secrets on the authorized device, they should not be able to successfully authenticate or decrypt as they are not physically in the home. We note that this threat model exceeds that of many secure protocols, which assume a malicious network but a trusted, secure end-user device.

4.3.3.2 Local Network Adversaries

We consider adversaries present on the local network. This can occur through compromising any number of the IoT devices on the network. This is a natural assumption as IoT devices have a history of vulnerabilities [62, 188]. The adversary can be remote, or have physical access to the devices. The latter models threats from other residents of the smart home, such as a malicious roommate or house-guest. In either case, the local network adversary will also be able to see all of the traffic over the LAN between the devices and the authorized device. They would also have the ability to use this access to perform denial-of-service. Despite all this, as long as the user's authorized device remains secure, the local network adversary should not be able to successfully execute the protocol.

These two adversaries represent the primary ways that a malicious party would try to undermine an at-home cryptographic system. Building a system robust against these two adversaries ensures that in-home compromise of the IoT devices or out-of-

home compromise of the user's authorized device does not violate the location-binding property of the system. We design for this threat model in a modular way, demonstrating the security of our system against each adversary independently. We assume that these adversaries do not collude, as compelled access adversaries are not assumed to have the capability to access personal devices besides those physically available to them [185, 210, 251]. However, in Section 4.4.3 we describe some extensions that would allow for our system to handle colluding adversaries as well.

4.4 SocIoTy

We are now ready to describe SocIoTy, our at-home cryptographic solution.

4.4.1 Preliminaries

We discuss SocIoTy in terms of its components:

- *Authorized device/authorized smartphone*: This device has reasonably good computational power and is carried by the user. We assume the authorized device is honest while within the home, but might be corrupted (*e.g.*, stolen or forcibly removed from the user) upon leaving the home. We assume the device supports *effaceable storage*, *i.e.*, allows for secure deletion of cryptographic secrets. Such functionality is common on modern smartphones [251].
- *Remote service*: The remote service is an Internet-accessible service with which the user wishes to interact through their at-home cryptography. In the authentication case, this is a service requiring login with two-factor authentication enabled. In the encryption case, this is a cloud storage endpoint.
- *IoT devices*: The user selects IoT devices from their smart home to participate in SocIoTy. The user selects any device with sufficient hardware and network capabilities to execute the protocol, which may include microcontroller-class devices.

4.4.1.1 Choosing the Correct Cryptographic Primitive

For both at-home authentication and encryption, we need to tie cryptographic operations to a particular location. One natural way to do this is to have IoT devices use a generic MPC protocol to perform both encryption and authentication, where the respective keys have been secret shared among all parties and the output is given directly to the smartphone. Unfortunately, generic MPC is too inefficient for our setting, involving multiple rounds of communication and expensive computation operations [115, 116, 218]. Particularly in IoT environments, where even RAM is significantly limited, we cannot use many standard tricks to improve performance and even hundreds of milliseconds per layer may introduce unacceptable latency.

We would instead prefer to have the smart home implement a single cryptographic primitive that is well suited for use in a wide range of applications. One primitive that could work is a PRF, which has standard transformations to both symmetric encryption schemes and MACs. The distributed version of a PRF that makes most sense in our setting is a TDPRF. As discussed in Section 4.2, a TDPRF allows $\geq t$ parties to compute partial evaluations of the PRF that can later be combined to recover the full PRF output, but to any group of $< t$ parties, the output of the TDPRF is indistinguishable from random. This helps with both security and availability: not every IoT device needs to be online to evaluate the TDPRF but any adversary that only compromises $< t$ devices cannot recover the correct output of the TDPRF on any point that they have not already seen.

4.4.1.2 Layering Security

While a TDPRF achieves some security against an adversary corrupting $< t$ parties, we would also like to handle the case where an adversary corrupts over this threshold, potentially even up to all the IoT devices in the home. To protect against such adversaries, the phone will also contribute to constructing correct output. In short, we will construct a new PRF P' from the proposed TDPRF of the smart home and a PRF P, with the same

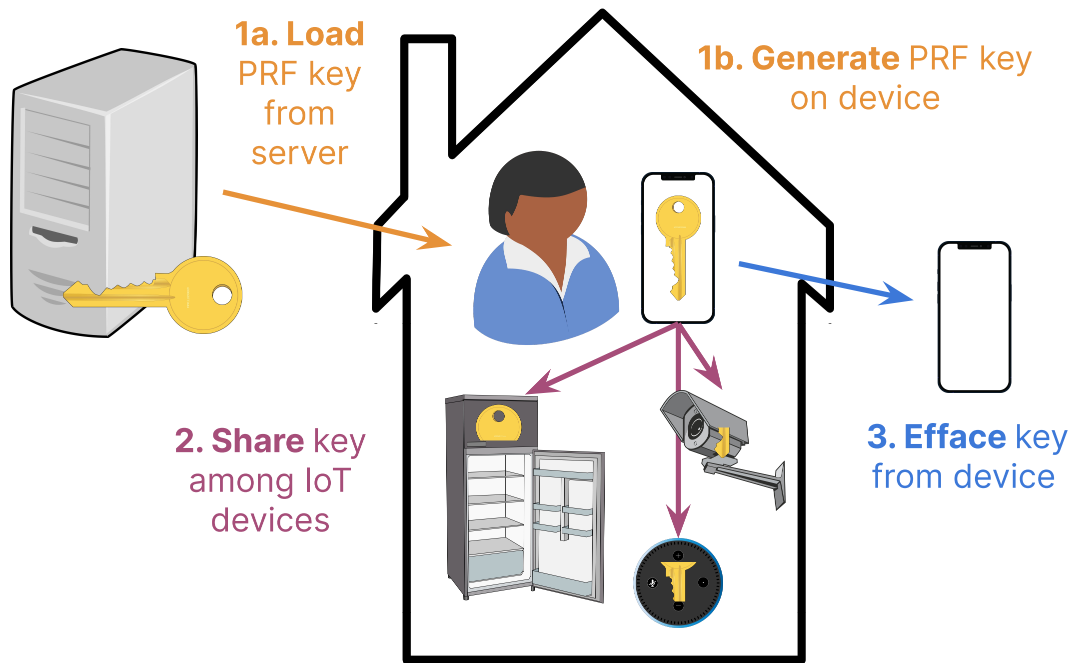


Figure 4.3. The Setup workflow of SocIoTy

co-domain as the TDPRF. If the composition of the TDPRF and P is pseudorandom, even when either of the TDPRF key or the key for P is leaked (and the smartphone is the only party who holds the key for P) the output of P' will appear pseudo-random to all adversaries covered in our model. This layering will also being more practically efficient to compute than any generic solution that only protects against a limited number of IoT device corruptions.

4.4.2 Protocol Description

We now briefly describe the normal operation of SocIoTy at a high level before describing the protocol in depth. When an authorized smartphone wants to register a new service with the smart home, it first generates the key material needed for itself and the home using a Setup algorithm (Figure 4.3). It gives the correct key shares to all the devices in the smart home and securely deletes them from its memory. When the smartphone later uses the service, it broadcasts over the LAN a request for a TDPRF evaluation. The

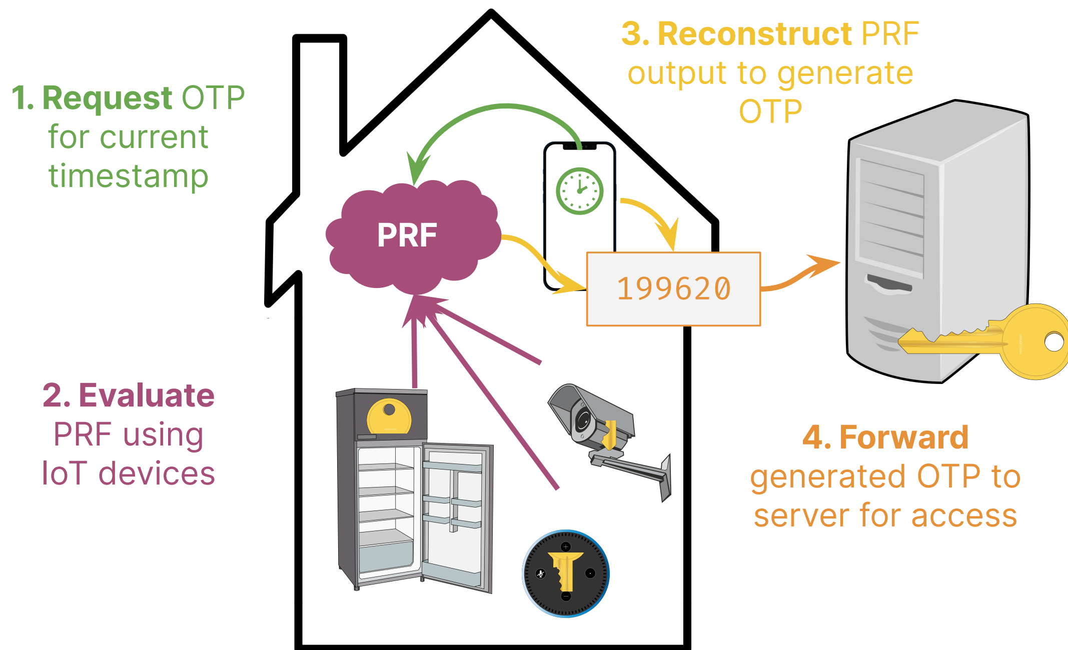


Figure 4.4. The Authentication workflow of SocIoTy.

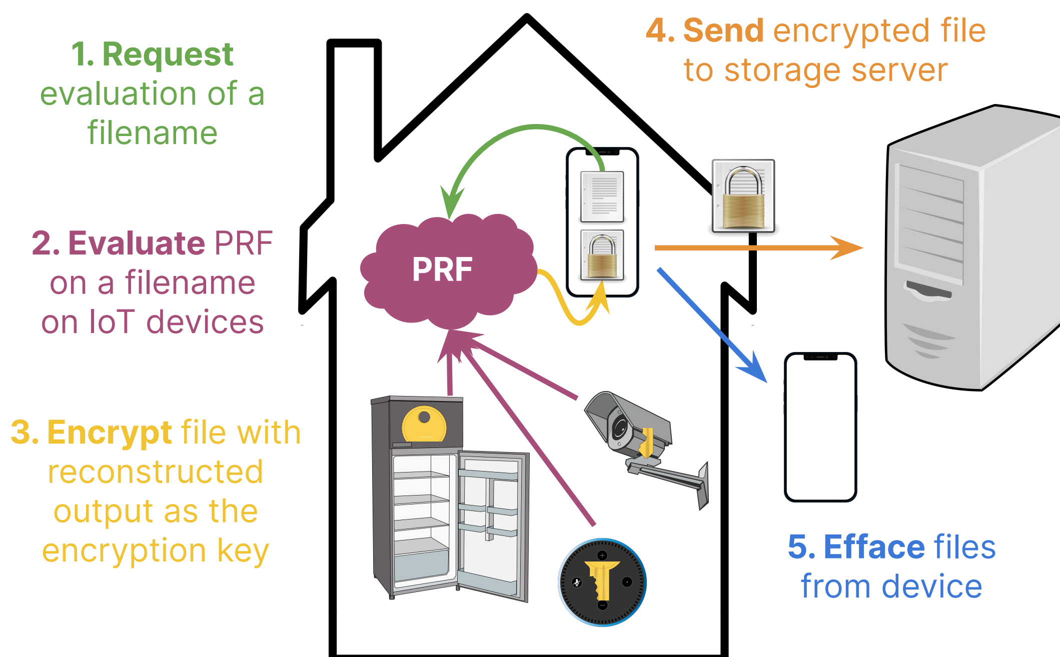


Figure 4.5. The Encryption workflow of SocIoTy.

phone waits until it receives at least t evaluation responses from the IoT devices before attempting reconstruction. Once reconstruction is completed, depending on whether the application is Authentication (Figure 4.4) or Encryption (Figure 4.5), the phone takes a series of actions. Any sensitive information is securely erased from the phone after the operation completes. What follows is a complete description of the Setup, Authentication and Encryption algorithms.

4.4.2.1 Setup

Let n be the number of smart devices a user owns and let t be a fixed number, equal to the number of devices expected to be online at any given point in time. The setup procedure is designed to produce two keys: one for the smartphone denoted by k_p and one split among the networked IoT devices denoted by K. In the case of authentication, the keys k_p and K will be provided by the remote service the phone is authenticating to. For encryption, k_p and K should be generated by the smartphone. The phone uses the $\mathsf{TDPRF.Gen}$ algorithm to share the key K as $K_1 \ldots K_n$. The key share K_i is given to device i. The phone then stores k_p and after sharing the shares of K, securely deletes all key material related to K. When a new IoT device is bought or sold from the smart home, a phone repeats this procedure, replacing k_p and K with new keys. We assume that in the case of authentication, the remote service provides a mechanism by which the symmetric TOTP key can be updated. Figure 4.3 illustrates the setup process.

4.4.2.2 Authentication

For authentication, the smartphone calculates a counter δ based on the current timestamp. It then sends an authentication request to all devices within the smart home. Each device with available bandwidth runs $\mathsf{TDPRF.PartialEval}(K_i, \delta)$ to get y_i and sends the resulting y_i to the phone. Once the phone has received t partial evaluations it recovers the PRF output and calculates its own PRF value. The two are then combined and the output is

Algorithm 4.1: SocIoTy Authentication

Input: k_p the key of the smartphone, δ a counter value derived from a timestamp
Output: TOTP token
Request smart home devices invoke $\mathsf{PartialEval}$ on δ and receive $\{y_i\}_{i \in T}$ where $T \subseteq [n], |T| \geq t$
$y \leftarrow \mathsf{TDPRF.Recon}(\{y_i\}_{i \in T})$
$z = y + \mathsf{PRF.Eval}(k_p, \delta)$
Output $z \pmod{10^6}$

Algorithm 4.2: SocIoTy Encryption

Input: k_p the key of the smartphone, m the name of the file, f the content of the file itself
Output: Encrypted file c
Request smart home devices invoke $\mathsf{PartialEval}$ on m and receive $\{y_i\}_{i \in T}$ where $T \subseteq [n], |T| \geq t$
$y \leftarrow \mathsf{TDPRF.Recon}(\{y_i\}_{i \in T})$
$z = y + \mathsf{PRF.Eval}(k_p, m)$
$k \leftarrow \mathsf{KDF.Derive}(z)$
$c \leftarrow \mathsf{AE.Encrypt}(k, f)$
Securely delete k
Output c

truncated to 6 ten digit numbers that are displayed to the smart phone user. The remote service can use k_p and K, along with the PRF.Eval and TDPRF.Eval algorithms, to check for correctness. An overview diagram is provided in Figure 4.4, with a more specific description in Algorithm 4.1.

4.4.2.3 Encryption

To encrypt and decrypt sensitive files, the smartphone first makes a request for an PRF evaluation on a filename m. The smart home devices conduct a partial PRF evaluation as $\mathsf{TDPRF.PartialEval}(K_i, m)$. The output of these evaluations is then given to the smartphone. The phone can then reconstruct the PRF output before combining it with the output of its own PRF evaluation on m using key k_p. The resulting PRF output is then used as an entropy source for a key derivation function, KDF [123]. The value output by KDF is a pseudorandom key which can then be used in an authenticated encryption scheme AE

to either encrypt or decrypt the file while providing strong confidentiality and integrity. After the operation is completed, the phone securely deletes the reconstructed key and potentially the plaintext file. This workflow is depicted in Figure 4.5 and described in Algorithm 4.2.

4.4.3 Security Analysis

We now give a justification of security for our construction against the relevant adversaries. Recall that we are concerned with two types of attackers (1) a compelled access adversary who may compromise the phone while it is abroad, but does not simultaneously have access to any device in the smart home and (2) a local network adversary that has direct physical access to IoT devices and any traffic over the LAN but cannot compromise the smart phone. We note that in the multi-user setting, other users are equivalent to adversary 2. To give a brief summary, security holds because of how the PRF and TDPRF are composed. Even if an adversary has access to one of k_p (adversary 1) or K (adversary 2), the total output retains PRF security and is indistinguishable from uniform. This means an adversary has no chance better than random of guessing either the TOTP value or the key used to encrypt files.

4.4.3.1 Extensions

We now discuss some special considerations for other types of network attacks and more powerful adversaries.

An local network adversary in practice has some slightly stronger adversarial capabilities, due to its ability to modify traffic. SocIoTy does not necessarily require authentication and encryption of home requests, as the security of the system relies on the dual-layered PRF. However, we do open up users to denial-of-service attacks on each of the relevant services, as an adversary could interfere with partial evaluations. In the case of authentication and file decryption, such an attack can only temporarily prevent correct functioning

of the system. More dire is the case of initializing user account values and file encryption, in which denial of service attacks may be unrecoverable and lead to a breakdown of system properties (*i.e.*, encrypting a file with a corrupted key and then deleting the plaintext). To protect against these attacks we can add authenticity checks to values. We add the recently standardized Ascon AEAD scheme [153] to our implementation to maintain security against these types of attacks, and evaluate its performance in Section 4.5.

We consider a more powerful access adversary who can gain control of both the phone and even one IoT device on the home network to be out of scope. We believe, for most use cases, this is a realistic assumption: even gaining the public-facing IP address of devices on the network is not something a foreign nation can do easily, without help from the user's local ISP. For those highly-targeted users for whom such an adversary could be realistic, though, turning off the smart home entirely when they leave the house will prevent this attack, giving the user the same security guarantees as an offline solution.

4.4.4 Deployment Flexibility

SocIoTy's design allows for significant flexibility when deploying on a smart home. We discuss these considerations in the paragraphs below.

4.4.4.1 Devices to Use

We envision SocIoTy as running on essentially any IoT device that has some form of networking capability, as the number and types of device vary from smart home to smart home. Users should try to use their more powerful devices to increase performance, but we believe this is not strictly necessary. We evaluate these performance claims in Section 4.5, using a wide range of devices to benchmark the SocIoTy protocol.

4.4.4.2 Multi-User Smart Homes

SocIoTy can support multiple users, each with their own services. Each device i holds a separate key K_i (for PartialEval) for each pair (u, s) describing a user u and service s. The wrong user u' cannot authenticate to s as u because they do not have the key k_p on u's smartphone. Thus, SocIoTy supports as many users and services as there is space for keys on the IoT devices. Similarly, SocIoTy also supports *multi-owner* setups, where the devices are not all owned by a single user. This is common in smart home settings, as devices can belong to roommates, landlords, or caretakers, to name a few. If all device owners cooperate, SocIoTy proceeds as normal. If owners deviate from the protocol, the worst that can happen is denial-of-service—not a security break.

4.4.4.3 Network Structure

Our design does not require a specific structure of the smart home network. Traditionally, protocols are designed point-to-point, where each device is able to directly communicate with each other device. For some smart homes, computation is handled through a hub, which acts as an intermediary for messages to and from the smart home devices, especially low-resource ones. SocIoTy is able to handle this case, which we investigate end-to-end in Section 4.5.4.

SocIoTy makes no liveness assumptions on the whole network. Other approaches, like generic multi-party computation [20, 44, 87, 239], would require all of the IoT devices to communicate with each other during the whole protocol. SocIoTy only needs each node to be active for one PartialEval. So a device can respond to a request, and go back to attending to its primary task (or return to sleep), without waiting for all of the other nodes to respond or for the final reconstruction to occur. Moreover, because our cryptographic protocol only requires one round of communication, we can also tolerate networks with very low available bandwidth.

4.4.4.4 Server Interface

SocIoTy meets our deployment goal of not requiring changes to the user interface, but we briefly discuss how SocIoTy impacts the *remote service*. When applying SocIoTy to encryption, the cloud server that provides storage does not change its interface. From its perspective, the user is still uploading a file: a SocIoTy-encrypted blob rather than a cleartext one. The cloud service stores it as it would any other file.

For authentication, however, the situation is different. The TOTP standard [148] recommends HMAC-SHA-1 as the underlying PRF. Our construction is not backwards-compatible with HMAC-SHA-1 in implementation, but the interface is the same: a call to TOTP returns a one-time password. Rather than using HMAC-SHA-1, a call to $\mathsf{TOTP}(sk, ts)$ in SocIoTy would instead invoke Algorithm 4.1, with the server keeping $sk = (k_p, K)$. We argue that this change is minimal, as the TOTP standard has a high level of abstraction [148]. Moreover, services are incentivized to make this change, as the additional security and flexibility of SocIoTy is a marketable benefit.

4.4.5 Instantiating the TDPRF

We must instantiate the TDPRF underlying SocIoTy's operations to deploy our solution in practice. We follow the decisional Diffie-Hellman-based construction of [149] for our TDPRF, using elliptic curve groups because of their efficiency in implementation. As is common when discussing elliptic curves, we use additive notation for group operations. Let G be a generator of an elliptic curve group of prime order p. We describe below the algorithms for our TDPRF $\mathsf{Gen}, \mathsf{PartialEval}$, and Recon, as well as the extra algorithm Eval (useful for a server implementation):

- $\mathsf{Gen}(1^\lambda, K, t, n)$: Sample a random polynomial f of degree $t - 1$ by uniformly sampling its coefficients from $\mathbb{Z}_p$, subject to the constraint that $f(0) = K$. The output party shares are the scalars $k_1 = f(1), k_2 = f(2), \ldots, k_n = f(n)$.

- $\mathsf{PartialEval}(K_i, x)$: Hash the input x uniformly onto a point P along the elliptic curve. Then, the output is simply $y_i \leftarrow K_i \cdot P$: scalar multiplication of the key *share* to the input point.

- $\mathsf{Recon}(\{y_i\}_{i \in Y})$: Let $\alpha_1 \ldots \alpha_t$ be the identities of the parties providing points $y_1 \ldots y_t$ to Recon. Consider the following function, defined $\forall i \in [t]$:

$$L_i(x) = \prod_{\forall j \neq i, j \in [t]} \frac{x - \alpha_j}{\alpha_i - \alpha_j}$$

  It is well known that given t points along a polynomial f, evaluation can be done at any point α as $f(\alpha) = \sum_{i=0}^{t} f(\alpha_i) \cdot L_i(\alpha)$. Given these points "in the exponent" it is possible to recover K "in the exponent". To be precise, we can recover the PRF output as:

$$y = \sum_{i=1}^{z} L_i(0) \cdot y_i = \sum_{i=1}^{z} L_i(0) \cdot f(\alpha_i) \cdot P = K \cdot P$$

- $\mathsf{Eval}(K, x)$: Hash the input x uniformly onto a point P along the elliptic curve. Then, the output is $y \leftarrow K \cdot P$: scalar multiplication of the *reconstructed* key to the input point.

Note that the hashing of the input x is important, as the output of this TDPRF is uniform only if its input is also uniform. If we model this hash function as a random oracle, security holds [149].

4.5 Evaluation

We now demonstrate the feasibility of our constructions on real IoT hardware.

4.5.1 Implementation

We implement SocIoTy in Rust due to its memory safety guarantees as well as its good platform support for IoT architectures. Additionally, we use the Curve25519 as our elliptic

Table 4.1. Hardware specifications of test devices as well as examples of comparable IoT devices.

Test Device	CPU	RAM	Comparable IoT Device
RPi 3B+	ARM Cortex-A53	1 GiB	Apple TV HD [12]
RPi 2B	ARM Cortex-A7	1 GiB	Amazon Echo Dot (3rd Gen) [25]
RPi Zero W	ARM1176JZF-S	512 MiB	Google Nest Thermostat E [135]
ESP32	Xtensa LX6	320 KiB	Belkin WeMo Light Switch [206]

curve and Ascon, the winner of the NIST lightweight cryptography competition [153], for authenticated encryption in our implementation. We will open-source all of our SocIoTy software and benchmarks for public use and review upon publication.

4.5.2 Microbenchmarks

We wish to understand how SocIoTy runs on a variety of devices. We performed our benchmarks on the following devices: 6 Raspberry Pi (RPi) Model 3B+ single-board computers (SBCs), 3 RPi Model 2B SBCs, 3 RPi Zero W SBCs, and 5 ESP32 microcontrollers. Raspberry Pis are increasingly being used a benchmarking platforms to simulate smart home devices in lieu of commercial devices; IoT device vendors do not support running arbitrary software for security reasons, limiting the ability to use them for development. Table 4.1 maps our test bed devices to comparable smart home devices.

Recent generations of Raspberry Pis have been increasing in computing power with specifications of up to 8GB of memory. Thus, we used both lower-end Raspberry Pis and smaller microcontrollers as representative devices to better simulate a network of heterogeneous IoT devices.

We first begin by presenting microbenchmark results for the algorithms of a TDPRF: (Gen, $\mathsf{PartialEval}$, Recon). All of our selected devices are capable of computing all three of these algorithms.

Table 4.2. Average runtimes for an evaluation of **PartialEval**, both without and with authenticated encryption (**AE**). All times are in milliseconds.

Experiment	RPi 3B+	RPi 2B	RPi Zero	ESP32
PartialEval	1.34	2.19	2.90	43.68
PartialEval (AE)	1.53	2.43	3.28	47.22

4.5.2.1 PartialEval

Because **PartialEval** will be conducted on resource-constrained IoT devices, microbenchmarks for it are very informative. As discussed in Section 4.4.5, each **PartialEval** in our implementation is one elliptic curve multiplication. We evaluate the performance of **PartialEval**, and of **PartialEval** followed by an authenticated encryption of the result using Ascon (denoted **AE**). The Raspberry Pis performed each task 100,000 times, and the ESP32 performed each 1,000 times, with the results in Table 4.2. The Raspberry Pis complete the task very quickly—less than 5 milliseconds on average. The sub-50ms average time on the ESP32s is also very promising; while an order of magnitude slower than the Raspberry Pis, this result shows that adding SocIoTy on even highly constrained devices will not induce noticeable latency. We also note that the overhead of authenticated encryption is minimal, even on the ESP32. As such, for the rest of our benchmarks, we have all nodes use **PartialEval** with Ascon to add security against network tampering (as discussed in Section 4.4.3).

4.5.2.2 Gen and Recon

We also perform microbenchmarks on **Gen** and **Recon**, and present our results in Figures 4.6 and 4.7. Each Raspberry Pi once again ran each task 100,000 times, with varying configurations of the total number of parties n and the threshold required to reconstruct t. **Gen** in our implementation only samples random values for the keys and, while the time to run does scale with each (n, t) pair, it operates on the order of several hundred microseconds (μs) on average. **Recon** takes longer, likely owing to the multiple elliptic

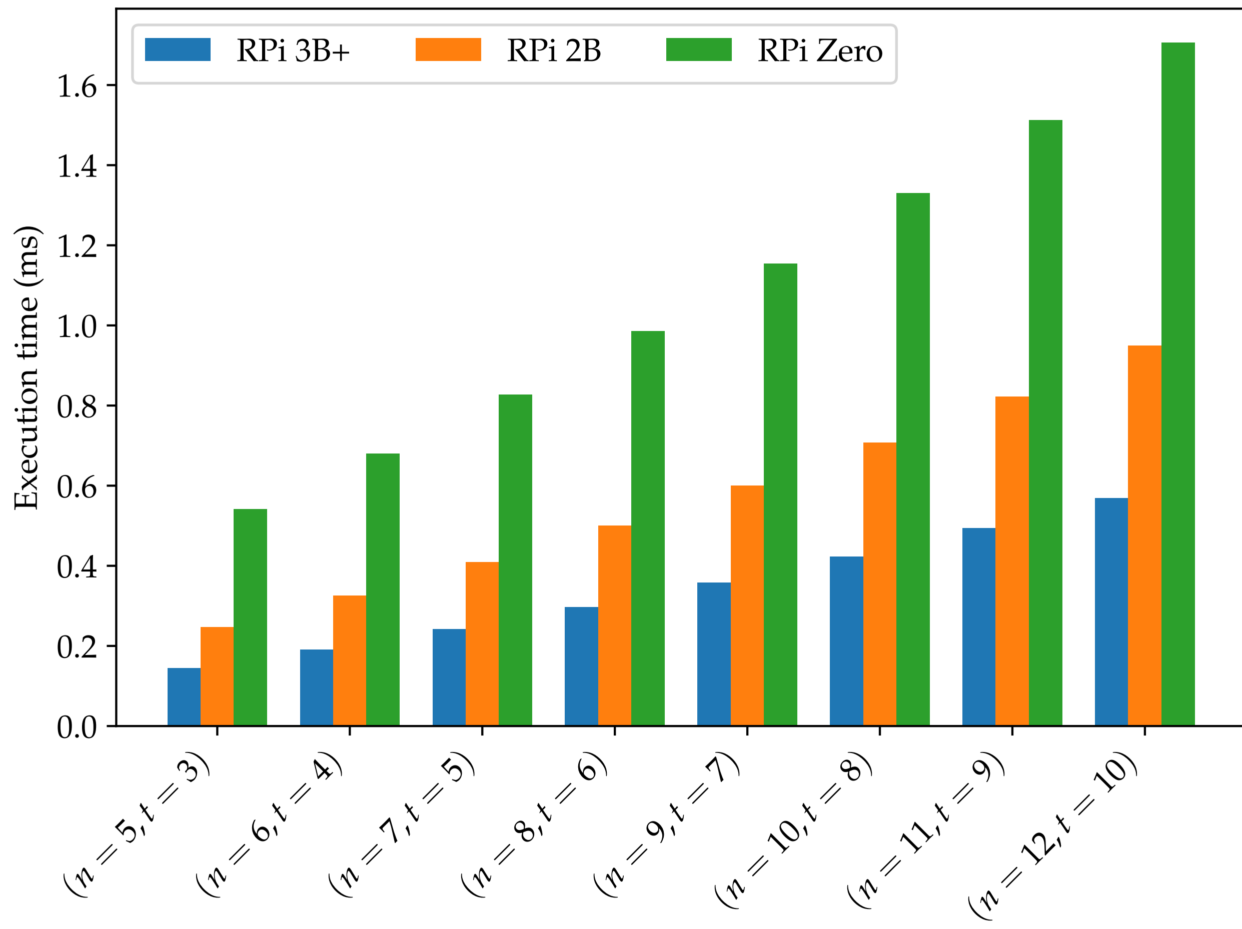


Figure 4.6. Microbenchmarks for Gen on different test devices over varying configurations of total number of parties n and reconstruction threshold t.

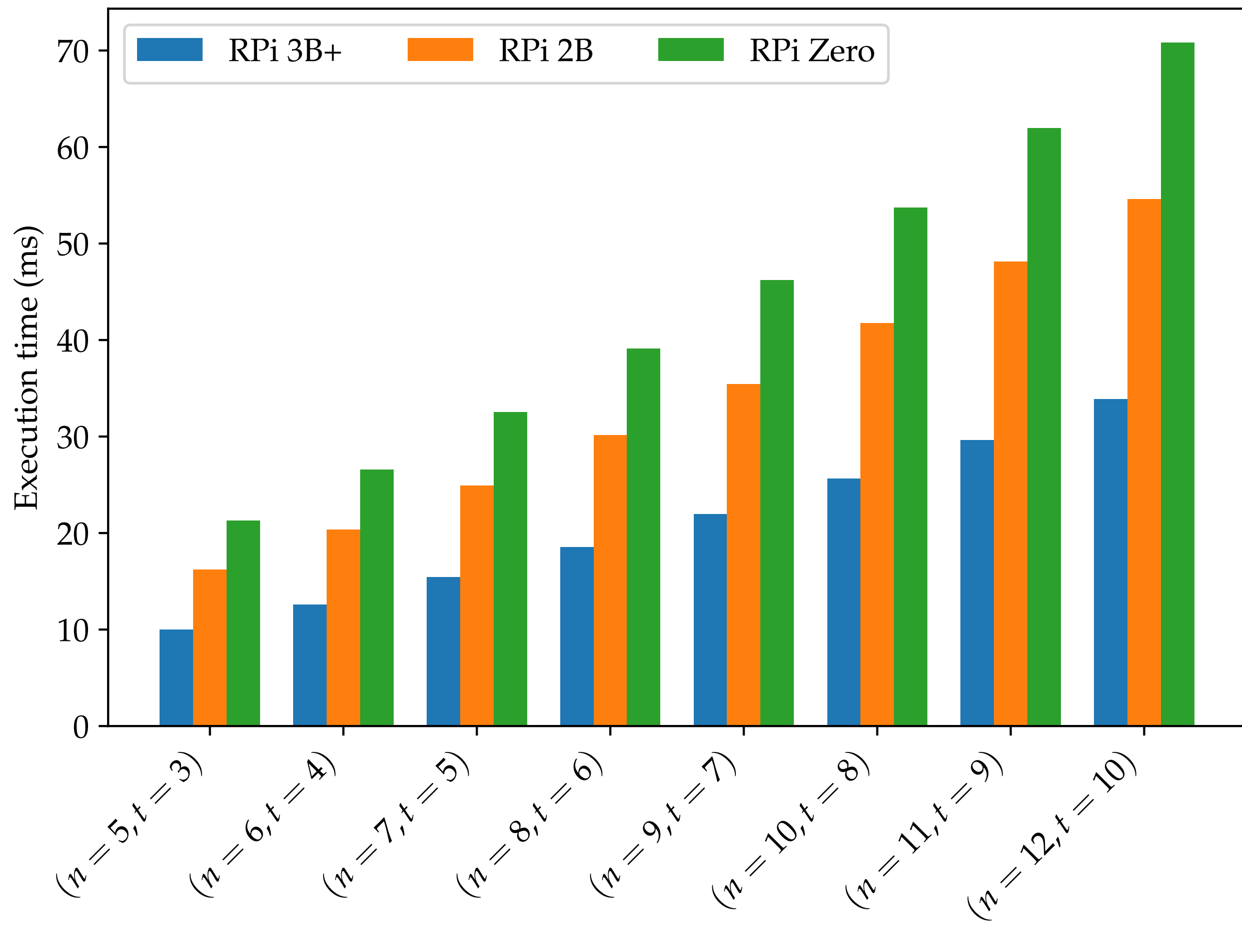


Figure 4.7. Microbenchmarks for Recon on different test devices over varying configurations of total number of parties n and reconstruction threshold t.

curve operations required to interpolate the partial evaluations and recover the PRF output. It similarly increases as the network grows, but even in a 12-device network, it only takes around 70 milliseconds on a Raspberry Pi Zero.

We believe that these microbenchmarks represent an upper bound on the execution time; as discussed in Section 4.4.1, we expect users to use their smartphone as the authorized device for Gen and Recon, and modern smartphones have much better processors than the ARM1176JZF-S found in the Pi Zero. While we do not envision users generating and recovering on even smaller, microcontroller-class devices, for completeness we evaluated how Gen and Recon fare on the ESP32 for different configurations of (n, t). These results can be found in Table 4.3.

4.5.3 Scalability Benchmarks

Our next set of experiments measures how the execution time of the evaluation of SocIoTy's TDPRF scales once communication between devices is involved. We set up two types of nodes, a request node and n evaluation nodes. The evaluation nodes are the Raspberry Pis: 6 RPi 3B+s (used for all benchmarks), 3 RPi 2Bs (used for $n \geq 7$), and 3 RPi Zeros (used for $n \geq 10$). In each run, the request node connects to all n evaluation nodes and makes a request for a timestamp δ. Each evaluation node then responds with the (authenticated-encrypted) PartialEval for δ, and the request node performs Recon once it has received (and decrypted) t responses. All of the nodes are on the same Wi-Fi network, and communication occurs over the Constrained Application Protocol (CoAP) [244], a popular point-to-point protocol in IoT.

We perform 1,000 of these runs for varying configurations of (n, t), and plot our results in Figure 4.8, with all variations of (n, t) in Table 4.4. Clearly, as the required threshold to reconstruct t increases, the execution time increases: more responses need to arrive. We also see a relatively large jump in average execution time at $n = 7$ and $n = 10$, likely because of the involvement of the less-powerful Pi 2Bs and Pi Zeros at each step.

Table 4.3. Microbenchmarks for Gen and Recon on the ESP32 devices over varying configurations of total number of parties n and reconstruction threshold t. All times are in milliseconds. Although we do not expect users to use the ESP32 for Gen or Recon, our implementation is nonetheless efficient enough for this purpose.

Configuration	ESP32 Gen	ESP32 Recon
$(n = 5, t = 3)$	41.16	382.47
$(n = 5, t = 4)$	41.63	382.47
$(n = 5, t = 5)$	42.12	382.11
$(n = 6, t = 4)$	49.78	495.02
$(n = 6, t = 5)$	50.33	495.02
$(n = 6, t = 6)$	50.89	494.48
$(n = 7, t = 5)$	58.55	622.77
$(n = 7, t = 6)$	59.18	622.77
$(n = 7, t = 7)$	59.83	622.02
$(n = 8, t = 6)$	67.48	765.77
$(n = 8, t = 7)$	68.20	765.77
$(n = 8, t = 8)$	68.92	764.76
$(n = 9, t = 7)$	76.58	923.98
$(n = 9, t = 8)$	77.36	923.98
$(n = 9, t = 9)$	78.17	922.68
$(n = 10, t = 8)$	85.83	1097.46
$(n = 10, t = 9)$	86.70	1097.46
$(n = 10, t = 10)$	87.73	1097.44
$(n = 11, t = 9)$	95.25	1286.10
$(n = 11, t = 10)$	96.18	1286.51
$(n = 11, t = 11)$	97.31	1286.09
$(n = 12, t = 10)$	104.81	1490.53
$(n = 12, t = 11)$	105.83	1490.53
$(n = 12, t = 12)$	107.05	1490.06

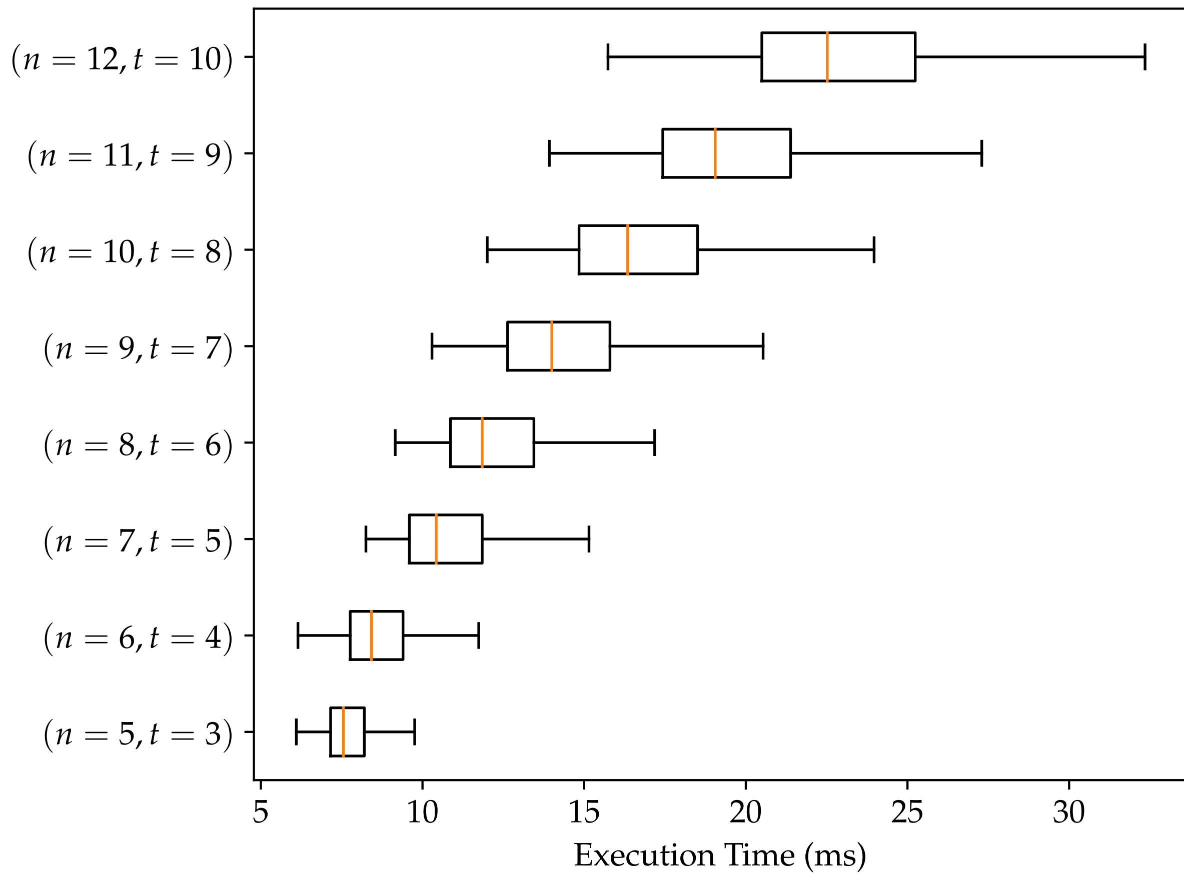


Figure 4.8. Protocol execution time over CoAP for varying configurations of total number of parties n and reconstruction threshold t.

Table 4.4. Protocol execution time CoAP over all evaluated configurations of total number of parties n and reconstruction threshold t. All times are in milliseconds. The large spike in μ, σ at $n \geq 10, n = t$ is likely due to faulty hardware.

Configuration	μ	σ
$(n = 5, t = 3)$	7.90	1.23
$(n = 5, t = 4)$	8.96	1.76
$(n = 5, t = 5)$	10.77	4.34
$(n = 6, t = 4)$	8.85	1.76
$(n = 6, t = 5)$	10.65	2.26
$(n = 6, t = 6)$	12.05	2.16
$(n = 7, t = 5)$	11.05	2.51
$(n = 7, t = 6)$	12.57	2.65
$(n = 7, t = 7)$	14.41	3.21
$(n = 8, t = 6)$	12.39	2.38
$(n = 8, t = 7)$	14.31	2.74
$(n = 8, t = 8)$	16.06	2.99
$(n = 9, t = 7)$	14.42	2.51
$(n = 9, t = 8)$	16.70	3.47
$(n = 9, t = 9)$	18.37	4.03
$(n = 10, t = 8)$	17.07	4.28
$(n = 10, t = 9)$	19.74	4.21
$(n = 10, t = 10)$	49.57	151.41
$(n = 11, t = 9)$	19.81	3.90
$(n = 11, t = 10)$	22.35	3.97
$(n = 11, t = 11)$	48.05	134.40
$(n = 12, t = 10)$	23.34	4.87
$(n = 12, t = 11)$	24.63	5.01
$(n = 12, t = 12)$	53.62	145.06

Note that there is a sharp increase when the number of nodes is $n \geq 10$ and the threshold is $t = n$. In this case, we found that one of the Raspberry Pi Zeros performs significantly worse on network communication than all other devices, perhaps due to a manufacturing issue. Necessarily, when the threshold is sufficiently large to encompass the slowest of devices, the computation becomes bounded by the slowest performing devices.

Regardless, even in a relatively large configuration like $(n = 12, t = 10)$, though, each full run takes less than 25 milliseconds on average. Thus, SocIoTy TDPRF evaluations are able to scale well as the smart home network adds devices.

4.5.4 End-to-End Deployment

We now perform an end-to-end deployment of SocIoTy. We focus on the authentication process depicted in Section 4.4.2. The results for authentication will be applicable to encryption as well, as the core of the two algorithms is the same. The only difference is the actual authenticated encryption of a file, which has minimal overhead on smartphones.

An end-to-end authentication system must support the generation of 2FA OTPs. So we built a smartphone app, based on an open-source implementation [18], with the same interface as common 2FA apps. Our app performs all of the steps in the TDPRF evaluation—making the $\mathsf{PartialEval}$ requests and Reconing the responses—and takes the additional step of converting the output of Recon into a six digit OTP and displaying it to the user. A screenshot of our app can be found in Figure 4.10.

As discussed in Section 4.4.4, smaller single purpose devices may not directly connect to the Internet or other devices found within the smart home, but rather connect to a central more-powerful hub. This hub coordinates the flow of data of each device to and from parts of the smart home or Internet. A commonly used IoT protocol for this is MQTT, in which devices *subscribe* to *topics* to receive information and *publish* to them to send information, while a central broker sends published data to subscribers.

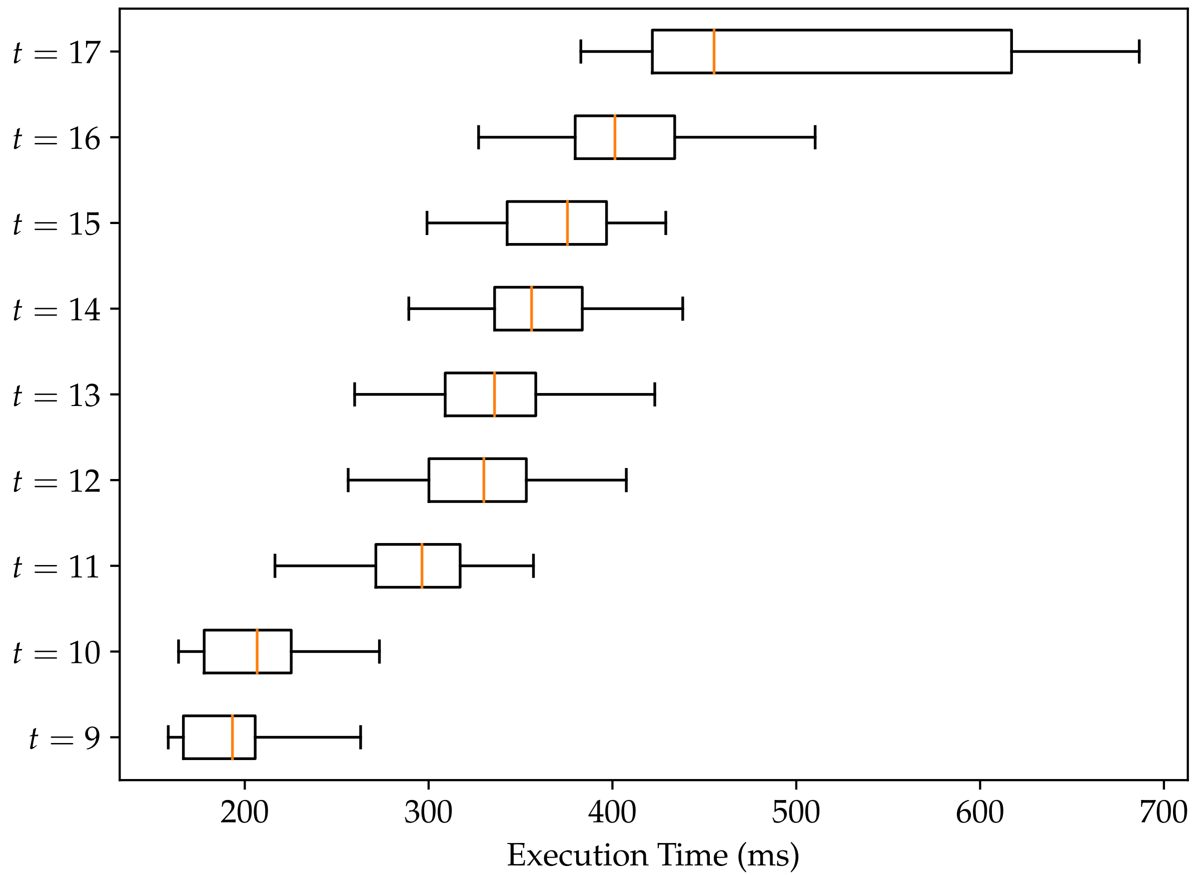


Figure 4.9. End-to-end OTP generation time using our iOS app for varying configurations of total number of parties n and reconstruction threshold t.

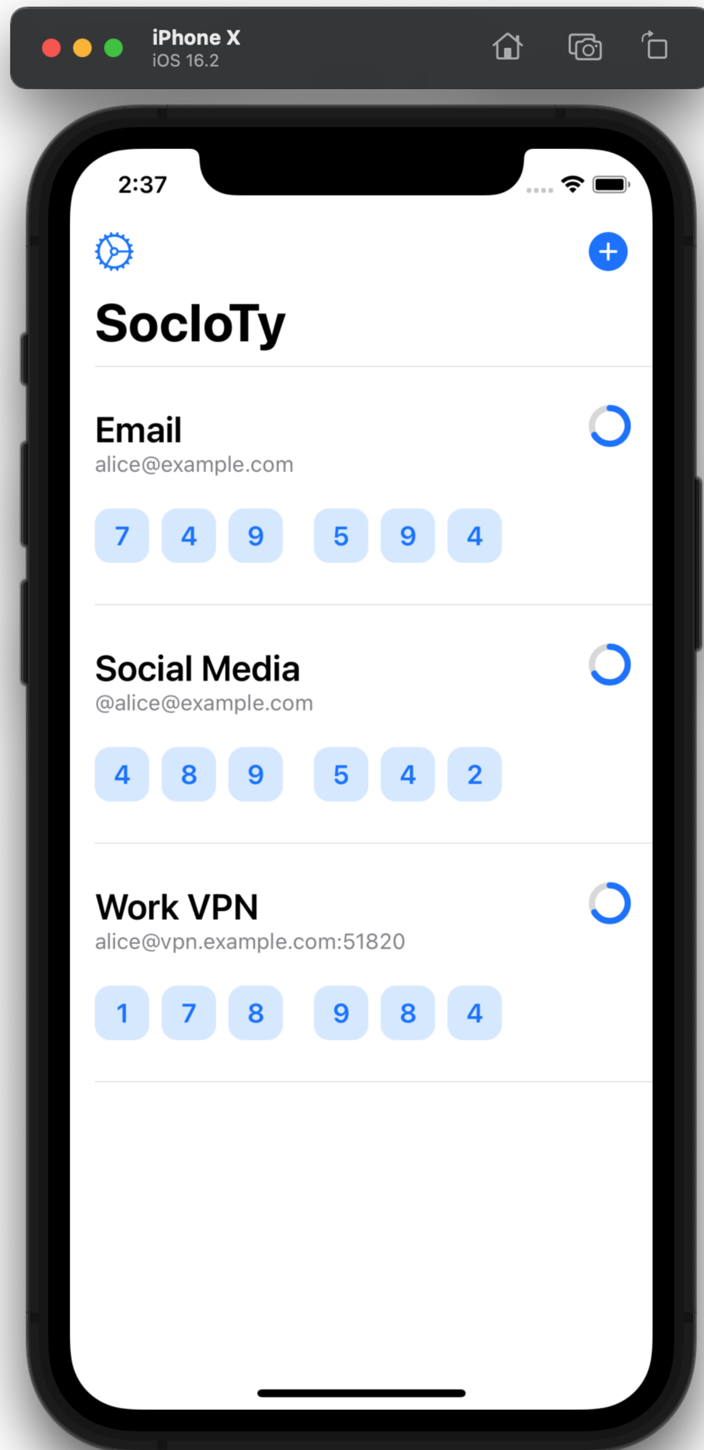


Figure 4.10. A Simulator screenshot of our iOS app. Note that all benchmarks were performed with a hardware iPhone X.

Keeping this in mind, we construct the following testbed to perform our end-to-end experiments. Our simulated smart home consists of the 12 Raspberry Pis from our experiment in Section 4.5.3, as well as 5 ESP32 microcontrollers—representing the class of devices that use lightweight IoT protocols like MQTT due to its hub architecture—for total of 17 evaluation nodes, all connected to the same Wi-Fi network. A 2018 iPhone X is used to run our smartphone app. We use a standard Ubuntu 21.04 server running on the same LAN as the MQTT broker.

Every tc seconds (represented by a full progress circle in Figure 4.10), the iOS app generates a new TOTP by doing the following:

1. The app calculates the TOTP counter value $\delta = \lfloor \frac{ts}{tc} \rfloor$ based on the current timestamp ts.

2. The app connects to the MQTT broker, subscribes to the MQTT topic `socioty/tdprf/`δ, and publishes δ to the topic `socioty/tdprf` to the broker.

3. Each node i is subscribed to `socioty/tdprf`, and receives δ from the broker.

4. Each node i then computes $y_i \leftarrow \mathsf{PartialEval}(K_i, \delta)$, and publishes it to `socioty/tdprf/`δ.

5. Once the app has t responses, it performs the remainder of Algorithm 4.1, reconstructing the output and displaying the new TOTP.

We consider the above steps one run, and we perform 100 runs, varying the threshold t while leaving the number of total devices fixed as $n = 17$.

We present our results of our end-to-end deployment benchmarks in Figure 4.9. We see that average execution times range from under 200ms at a majority threshold $t = 9$ to under 500ms when all devices are involved at $t = 17$. Similarly to our results in 4.5.3, we see a sharp increase in execution times as we rely more on weaker devices to provide their responses. The large spread at $t = 17$ is likely due to the app waiting for a straggler

device that receives the request last and computes a response last; after all, an n-of-n system will be as fast as its slowest component.

Our experiments show that we are able to request and reconstruct the OTP well within the lifetime of the TOTP, $tc = 30$ seconds. For a threshold set to a simple majority of devices the response is quick, accounting for less than 1% of the TOTP lifetime. We find these results demonstrate the practicality of our system to be used seamlessly as a TOTP generator.

4.6 Related Work

We now compare SocIoTy to other work with similar goals. We summarize our comparisons in Table 4.5.

4.6.1 Location-Based Cryptography

Heuristics around location-based cryptography were originally formed in the networking community, with a set of "geo-encryption" algorithms [6, 77, 166, 186] that introduce location and time as additional parameters to a cryptographic operation by using satellite data. More formal cryptographic definitions were introduced by Chandran et al. [41] as "position-based cryptography," wherein they demonstrate the impossibility of verifying the physical position (based on radio wave communications) of a number of colluding provers within a space in the standard model. Works since have explored the assumptions made by Chandran and their implications in complexity theory [32] and in the quantum setting [34].

Phuong et al. developed a location-based encryption scheme in 2019 [165]. However, their scheme requires bilinear maps (as used in an attribute-based encryption scheme) to achieve constant ciphertext size decryptable at arbitrary points within 2-D or 3-D grids. Further, they rely on time-specific encryption [113, 160] to ensure decryption only at

Table 4.5. A comparison of related work with similar goals to those of SocIoTy.

Category	Authentication?	Encryption?	Location-binding?	Uses existing hardware?	Suitable for non-experts?
Geo-encryption [6, 77, 166, 186]	✗	✓	✓	✗	✗
Position-based crypto [32, 34, 41]	✓	✓	✓	N/A	✗
Time-specific encryption [113, 160, 165]	✗	✓	✓	N/A	✗
HSMs [95, 105, 226]	✓	✓	✓	✗	✗
Wearable devices [36, 50, 193]	✓	✗	✗	✓	✓
Proximity measurement [9, 247]	✓	✗	✗	✓	✗
Pico [199]	✓	✗	✓	✗	✓
SocIoTy (*proposal*)	✓	✓	✓	✓	✓

particular points for a given ciphertext.

4.6.2 Hardware Security Modules (HSMs)

Hardware security modules are separate, dedicated computing devices that protect cryptographic keys by storing them and monitoring their access and usage. They provide tamper-evidence or even tamper-resistance through the use of special hardware. Once tampering is detected, the device may stop functioning properly or delete its secret keys. HSMs can be used to protect keys used by certificate authorities, banks, and cryptocurrency wallets. They are present within vehicles [226], operational technology [105], and clouds [95]. HSMs act as trusted security anchors and gateway to the network. They securely generate, store, and process security-critical material shielded from any potentially malicious actor on the network and outside of it.

While providing good security guarantees on paper, historically HSMs have been too expensive for average consumers at the highest security levels and therefore have limited usability outside of large corporations [107, 114]. A more modern approach to HSMs was explored in the form of portable password storage hardware that pairs with other devices and applications to exchange keys and enable further seamless authentication. Pico [199] is an example of such device that can be shaped as a watch, a key fob, a bracelet or an item of jewellery. Pico uses with Picosiblings to enable a more coordinated approach to password storing and usage with other Pico devices. Instead of creating dedicated hardware, SocIoTy uses existing IoT hardware that has completely different purpose to provide security properties.

4.6.3 Security via IoT Devices

Most works in the literature that use the properties of IoT for security focus on IoT devices themselves, either by enhancing their security or facilitating easier authentication. Zhang et al. [247] describe an easier authentication for IoT devices by gesturing with a

smartphone in close proximity to the devices. Aman et al. [9] used a similar concept for the authentication of IoT devices by accounting for physical location. These works do not provide location-binding for user data, however. Some works do employ IoT characteristics for user authentication; in particular, [36, 50, 193] use wearable IoT devices as a second-factor for authentication.

Chapter 5

Conclusion

In this work, we explore a suite of techniques designed to give users control over their everyday computing, extracting security and privacy guarantees in the process.

First, in Chapter 2, we present *DOVE*, which offers an approach to achieve data-oblivious computation within a TEE for programs originally written in languages with complex stacks such as R. The approach takes as input a high-level program and transforms it to an intermediate representation (DOT) that can be more easily reasoned about with respect to providing data obliviousness on a constrained TCB. This gives the advantage of being able to program in a familiar and convenient language while providing a very strong security guarantee. We demonstrate a design and implementation that can cover a significant range of programs with efficiency that is an acceptable trade-off for the benefits.

Next, in Chapter 3, we present an analysis of the practical limitations of using cryptographically secure steganography on real, useful distributions, identifying the need for samplers and impractical entropy requirements as key impediments. We show that adapting existing public key techniques is possible, but produces stegotext that are extremely inefficient. We then present *Meteor*, a novel symmetric key steganographic system that dramatically outperforms public key techniques by fluidly adapting to changes in entropy. We evaluate Meteor, implementing it on GPU, CPU, and mobile, showing that it

is an important first step for universal, censorship-resistant steganography. We compare Meteor to existing insecure steganographic techniques from the NLP literature, showing it has comparable performance while actually achieving cryptographic security.

Finally, in Chapter 4, we present *SocIoTy*, an at-home cryptography system designed with non-technical users in mind. SocIoTy allows users to bind their secrets to their smart homes, giving them the opportunity to opt-in to additional protections for sensitive tasks. We protect against strong classes of adversaries, while providing the functionalities users expect, like authentication and encryption. Our benchmarks show that SocIoTy is practical, efficient, and conducive to deployment on real smart homes. In the future, we plan exploring what other at-home services we can provide on top of IoT devices through systems like SocIoTy.

The solutions considered in this work can each be understood as one part of a better digital life. Users could use DOVE for their cloud-based data science work, communicate with their friends and families via Meteor on their smartphones, and live inside of a SocIoTy-enabled smart home for authentication and encryption. Taken as a unit, it is a compelling model for everyday computing that is secure against realistic threat models while remaining efficient on consumer hardware. This line of research proves that there is hope for strong, yet practical, security and privacy for the modern world.

References

[1] AI Writer. http://ai-writer.com/.

[2] Onur Aciicmez, Jean-Pierre Seifert, and Cetin Kaya Koc. Predicting Secret Keys via Branch Prediction. *IACR'06*.

[3] Thomas Agrikola, Geoffroy Couteau, Yuval Ishai, Stanislaw Jarecki, and Amit Sahai. On pseudorandom encodings. In Rafael Pass and Krzysztof Pietrzak, editors, *TCC 2020: 18th Theory of Cryptography Conference, Part III*, volume 12552 of *Lecture Notes in Computer Science*, pages 639–669, Durham, NC, USA, November 16–19, 2020. Springer, Heidelberg, Germany.

[4] Roee Aharoni, Moshe Koppel, and Yoav Goldberg. Automatic detection of machine translated text and translation quality estimation. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, volume 2, pages 289–295, 2014.

[5] Adil Ahmad, KyungTae Kim, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. Obliviate: A Data Oblivious Filesystem for Intel SGX. In *NDSS'18*.

[6] Ala Al-Fuqaha and Omar Al-Ibrahim. Geo-encryption protocol for mobile networks. *Computer Communications*, 30(11-12):2510–2517, 2007.

[7] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port contention for fun and profit. IACR'18.

[8] T Alves and D Felton. TrustZone: Integrated hardware and software security. *ARM white paper*, 2004.

[9] Muhammad Naveed Aman, Mohamed Haroon Basheer, and Biplab Sikdar. Two-factor authentication for iot with location information. *IEEE Internet of Things Journal*, 6(2):3335–3351, 2018.

[10] Ross J Anderson and Fabien AP Petitcolas. On the limits of steganography. *IEEE Journal on selected areas in communications*, 16(4):474–481, 1998.

[11] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. On subnormal floating point and abnormal timing. In *S&P'15*.

[12] Apple Inc. Apple tv hd technical specifications. https://support.apple.com/kb/SP724, May 2021. Accessed 2/27/2023.

[13] Arian Avalos, Hailin Pan, Cai Li, Jenny P Acevedo-Gonzalez, Gloria Rendon, Christopher J Fields, Patrick J Brown, Tugrul Giray, Gene E Robinson, Matthew E Hudson, et al. A soft selective sweep during rapid evolution of gentle behaviour in an Africanized honeybee. *Nature communications*, 8(1):1550, 2017.

[14] Michael Backes and Christian Cachin. Public-key steganography with active attacks. In Joe Kilian, editor, *TCC 2005: 2nd Theory of Cryptography Conference*, volume 3378 of *Lecture Notes in Computer Science*, pages 210–226, Cambridge, MA, USA, February 10–12, 2005. Springer, Heidelberg, Germany.

[15] Anton Bakhtin, Sam Gross, Myle Ott, Yuntian Deng, Marc'Aurelio Ranzato, and Arthur Szlam. Real or fake? learning to discriminate machine from human generated text, 2019.

[16] Shumeet Baluja. Hiding images in plain sight: Deep steganography. In *Neural Information Processing Systems*, 2017.

[17] Elaine Barker and John Kelsey. Nist special publication 800-90a revision 1 recommendation for random number generation using deterministic random bit generators, 2015.

[18] Bastian Jaansen. Authenticator. https://github.com/BastiaanJansen/

Authenticator, 2020.

[19] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. *ACM TOCS'15*.

[20] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *20th Annual ACM Symposium on Theory of Computing*, pages 1–10, Chicago, IL, USA, May 2–4, 1988. ACM Press.

[21] Sebastian Berndt and Maciej Liskiewicz. On the gold standard for security of universal steganography. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2018, Part I*, volume 10820 of *Lecture Notes in Computer Science*, pages 29–60, Tel Aviv, Israel, April 29 – May 3, 2018. Springer, Heidelberg, Germany.

[22] Daniel J. Bernstein. Curve25519: New Diffie-Hellman Speed Records. In *PKC'06*.

[23] Daniel J. Bernstein. The Poly1305-AES Message-Authentication Code. In *FSE'05*.

[24] Marc Bevand. My experience with the great firewall of china. http://blog.zorinaq.com/my-experience-with-the-great-firewall-of-china/, Jan 2016.

[25] B&H Photo. Amazon echo dot (3rd generation, charcoal). https://www.bhphotovideo.com/c/product/1437065-REG/amazon_b0792kthkj_echo_dot_3rd_generation.html/specs, 2023. Accessed 2/27/2023.

[26] Marina Blanton, Aaron Steele, and Mehrdad Alisagari. Data-oblivious graph algorithms for secure computation and outsourcing. In *ASIA CCS'13*.

[27] OpenAI Blog. Better language models and their implications. Available at https://openai.com/blog/better-language-models/, February 2019.

[28] Manuel Blum and Silvio Micali. How to generate cryptographically strong se-

quences of pseudo random bits. In *23rd Annual Symposium on Foundations of Computer Science*, pages 112–117, Chicago, Illinois, November 3–5, 1982. IEEE Computer Society Press.

[29] Kevin Bock, iyouport, Anonymous, Louis-Henri Merino, David Fifield, Amir Houmansadr, and Dave Levin. Exposing and circumventing china's censorship of esni, 8 2020.

[30] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. *CoRR'17*.

[31] T. Brennan, N. Rosner, and T. Bultan. JIT leaks: Inducing timing side channels through just-in-time compilation. In *S&P'20*.

[32] Joshua Brody, Stefan Dziembowski, Sebastian Faust, and Krzysztof Pietrzak. Position-based cryptography and multiparty communication complexity. In Yael Kalai and Leonid Reyzin, editors, *TCC 2017: 15th Theory of Cryptography Conference, Part I*, volume 10677 of *Lecture Notes in Computer Science*, pages 56–81, Baltimore, MD, USA, November 12–15, 2017. Springer, Heidelberg, Germany.

[33] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[34] Harry Buhrman, Nishanth Chandran, Serge Fehr, Ran Gelles, Vipul Goyal, Rafail Ostrovsky, and Christian Schaffner. Position-based quantum cryptography: Impossibility and constructions. In Phillip Rogaway, editor, *Advances in Cryptology*

– *CRYPTO 2011*, volume 6841 of *Lecture Notes in Computer Science*, pages 429–446, Santa Barbara, CA, USA, August 14–18, 2011. Springer, Heidelberg, Germany.

[35] Christian Cachin. An information-theoretic model for steganography. Cryptology ePrint Archive, Report 2000/028, 2000. https://eprint.iacr.org/2000/028.

[36] Yetong Cao, Qian Zhang, Fan Li, Song Yang, and Yu Wang. Ppgpass: Nonintrusive and secure mobile two-factor authentication via wearables. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, pages 1917–1926. IEEE, 2020.

[37] Sunjay Cauligi, Gary Soeller, Fraser Brown, Brian Johannesmeyer, Yunlu Huang, Ranjit Jhala, and Deian Stefan. Fact: A flexible, constant-time programming language. *SecDev'17*.

[38] Somnath Chakrabarti, Thomas Knauth, Dmitrii Kuvaiskii, Michael Steiner, and Mona Vij. Chapter 8 - trusted execution environment with intel sgx. In Xiaoqian Jiang and Haixu Tang, editors, *Responsible Genomic Data Sharing*, pages 161 – 190. Academic Press, 2020.

[39] Eva KF Chan. Handy R functions for genetics research. https://github.com/ekfchan/evachan.org-Rscripts, 2019.

[40] T-H. Hubert Chan, Yue Guo, Wei-Kai Lin, and Elaine Shi. Cache-oblivious and data-oblivious sorting and applications. *IACR'17*.

[41] Nishanth Chandran, Vipul Goyal, Ryan Moriarty, and Rafail Ostrovsky. Position based cryptography. In Shai Halevi, editor, *Advances in Cryptology – CRYPTO 2009*, volume 5677 of *Lecture Notes in Computer Science*, pages 391–407, Santa Barbara, CA, USA, August 16–20, 2009. Springer, Heidelberg, Germany.

[42] Ching-Yun Chang and Stephen Clark. Linguistic steganography using automatically generated paraphrases. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, HLT '10, pages 591–599, Stroudsburg, PA, USA, 2010. Association for Computational

Linguistics.

[43] Ching-Yun Chang and Stephen Clark. Practical linguistic steganography using contextual synonym substitution and a novel vertex coding method. *Computational Linguistics*, 40(2):403–448, Jun 2014.

[44] David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (abstract) (informal contribution). In Carl Pomerance, editor, *Advances in Cryptology – CRYPTO'87*, volume 293 of *Lecture Notes in Computer Science*, page 462, Santa Barbara, CA, USA, August 16–20, 1988. Springer, Heidelberg, Germany.

[45] Marc Chaumont. Deep learning in steganography and steganalysis from 2015 to 2018, 2019.

[46] Chia che Tsai, Donald E. Porter, and Mona Vij. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *USENIX ATC'17*.

[47] Feng Chen, Chenghong Wang, Wenrui Dai, Xiaoqian Jiang, Noman Mohammed, Md Momin Al Aziz, Md Nazmus Sadat, Cenk Sahinalp, Kristin Lauter, and Shuang Wang. PRESAGE: privacy-preserving genetic testing via software guard extension. *BMC medical genomics*, 10(2):48, 2017.

[48] Feng Chen, Shuang Wang, Xiaoqian Jiang, Sijie Ding, Yao Lu, Jihoon Kim, S Cenk Sahinalp, Chisato Shimizu, Jane C Burns, Victoria J Wright, et al. Princess: Privacy-protecting rare disease international network collaboration via encryption through software guard extensions. *Bioinformatics*, 33(6):871–878, 2016.

[49] Chrysta Cherrie. The 2021 state of the auth report: 2fa climbs, while password managers and biometrics trend. https://duo.com/blog/the-2021-state-of-the-auth-report-2fa-climbs-password-managers-biometr Sept 2021. Accessed 2022-07-29.

[50] John Chuang. One-step two-factor authentication with wearable bio-sensors. In

Symposium on Usable Privacy and Security-SOUPS, volume 14, 2014.

[51] Cisco. Configure anyconnect secure mobility client using one-time password (otp) for two-factor authentication on an asa. https://www.cisco.com/c/en/us/support/docs/security/anyconnect-secure-mobility-client/213931-configure-anyconnect-secure-mobility-cli.html, March 2020. Accessed 2022-07-29.

[52] Kate Conger. Whatsapp blocked in brazil again. https://techcrunch.com/2016/07/19/whatsapp-blocked-in-brazil-again/, Jul 2016.

[53] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *S&P'09*.

[54] Intel Corporation. Processor Counter Monitor. https://github.com/opcm/pcm.

[55] Marta Costa-Jussa and José Fonollosa. Character-based neural machine translation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, pages 357–361, 03 2016.

[56] Victor Costan and Srinivas Devadas. Intel SGX explained. IACR'16.

[57] Silas Cutler. Project 25499 ipv4 http scans. https://scans.io/study/mi.

[58] Dana Dachman-Soled, Georg Fuchsbauer, Payman Mohassel, and Adam O'Neill. Enhanced chosen-ciphertext security and applications. In Hugo Krawczyk, editor, *PKC 2014: 17th International Conference on Theory and Practice of Public Key Cryptography*, volume 8383 of *Lecture Notes in Computer Science*, pages 329–344, Buenos Aires, Argentina, March 26–28, 2014. Springer, Heidelberg, Germany.

[59] Falcon Dai and Zheng Cai. Towards near-imperceptible steganographic text. *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 2019.

[60] David Darais, Chang Liu, Ian Sweet, and Michael Hicks. A language for probabilistically oblivious computation. *CoRR'17*.

[61] Nenad Dedic, Gene Itkis, Leonid Reyzin, and Scott Russell. Upper and lower bounds on black-box steganography. In Joe Kilian, editor, *TCC 2005: 2nd Theory of Cryptography Conference*, volume 3378 of *Lecture Notes in Computer Science*, pages 227–244, Cambridge, MA, USA, February 10–12, 2005. Springer, Heidelberg, Germany.

[62] Jyoti Deogirikar and Amarsinh Vidhate. Security attacks in iot: A survey. In *2017 International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud)(I-SMAC)*, pages 32–37. IEEE, 2017.

[63] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 21–21, Berkeley, CA, USA, 2004. USENIX Association.

[64] Jack Doerner, David Evans, and abhi shelat. Secure stable matching at scale. *IACR'16*.

[65] Duo. Duo authentication for epic. https://duo.com/docs/epic, Oct 2021. Accessed 7/29/2022.

[66] Zakir Durumeric, Eric Wustrow, and J Alex Halderman. Zmap: Fast internet-wide scanning and its security applications. In *Presented as part of the 22nd USENIX Security Symposium USENIX Security 13)*, pages 605–620, 2013.

[67] Kevin P Dyer, Scott E Coull, Thomas Ristenpart, and Thomas Shrimpton. Protocol misidentification made easy with format-transforming encryption. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 61–72. ACM, 2013.

[68] Kevin P. Dyer, Scott E. Coull, Thomas Ristenpart, and Thomas Shrimpton. Protocol misidentification made easy with format-transforming encryption. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013: 20th Conference on*

Computer and Communications Security, pages 61–72, Berlin, Germany, November 4–8, 2013. ACM Press.

[69] Kevin P. Dyer, Scott E. Coull, and Thomas Shrimpton. Marionette: A programmable network traffic obfuscation system. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 367–382, Washington, D.C., 2015. USENIX Association.

[70] Roya Ensafi, David Fifield, Philipp Winter, Nick Feamster, Nicholas Weaver, and Vern Paxson. Examining how the great firewall discovers hidden circumvention servers. In *Proceedings of the 2015 Internet Measurement Conference*, pages 445–458, 2015.

[71] Roya Ensafi, Philipp Winter, Abdullah Mueen, and Jedidiah R. Crandall. Analyzing the great firewall of china over space and time. *Proceedings on Privacy Enhancing Technologies*, 2015(1):61–76, January 2015.

[72] Saba Eskandarian and Matei Zaharia. An oblivious general-purpose SQL database for the cloud. *CoRR'17*.

[73] Dmitry Evtyushkin and Dmitry Ponomarev. Covert channels through random number generator: Mechanisms, capacity estimation and mitigations. In *CCS '16*.

[74] Dmitry Evtyushkin, Ryan Riley, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Branchscope: A new side-channel attack on directional branch predictor. In *ASPLOS'18*.

[75] Tina Fang, Martin Jaggi, and Katerina Argyraki. Generating steganographic text with lstms. *Proceedings of ACL 2017, Student Research Workshop*, 2017.

[76] David Fifield, Chang Lan, Rod Hynes, Percy Wegmann, and Vern Paxson. Blocking-resistant communication through domain fronting. *Proceedings on Privacy Enhancing Technologies*, 2015(2):46–64, 2015.

[77] Mahdi Daghmechi Firoozjaei and Javad Vahidi. Implementing geo-encryption in gsm cellular network. In *2012 9th International Conference on Communications*

(COMM), pages 299–302. IEEE, 2012.

[78] Ben Fisch, Dhinakaran Vinayagamurthy, Dan Boneh, and Sergey Gorbunov. IRON: Functional encryption using intel SGX. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017: 24th Conference on Computer and Communications Security*, pages 765–782, Dallas, TX, USA, October 31 – November 2, 2017. ACM Press.

[79] Tom Fish. Whatsapp banned: Countries where whatsapp is blocked mapped. https://www.express.co.uk/life-style/science-technology/1166191/whatsapp-ban-map-which-countries-where-whatsapp-blocked-censorship-china-banned, Aug 2019.

[80] Freedom House. Freedom on the net 2018 map. https://freedomhouse.org/report/freedom-net/freedom-net-2018/map, 2018.

[81] Sergey Frolov, Fred Douglas, Will Scott, Allison McDonald, Benjamin VanderSloot, Rod Hynes, Adam Kruger, Michalis Kallitsis, David G Robinson, Steve Schultze, et al. An isp-scale deployment of tapdance. In *7th {USENIX} Workshop on Free and Open Communications on the Internet ({FOCI} 17)*, 2017.

[82] Sergey Frolov and Eric Wustrow. The use of tls in censorship circumvention. In *NDSS*, 2019.

[83] Xiaoyi Gao and Joshua Starmer. Human population structure detection via multilocus genotype clustering. *BMC genetics*, 8(1):34, 2007.

[84] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. IACR'16.

[85] Sebastian Gehrmann, Hendrik Strobelt, and Alexander M. Rush. Gltr: Statistical detection and visualization of generated text, 2019.

[86] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions (extended abstract). In *25th Annual Symposium on Foundations of Computer*

Science, pages 464–479, Singer Island, Florida, October 24–26, 1984. IEEE Computer Society Press.

[87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred Aho, editor, *19th Annual ACM Symposium on Theory of Computing*, pages 218–229, New York City, NY, USA, May 25–27, 1987. ACM Press.

[88] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on Intel SGX. In *EuroSec'17*.

[89] Andreas Graefe. Guide to automated journalism, 2016.

[90] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks. In *USENIX Security'18*.

[91] Johann Großschädl, Elisabeth Oswald, Dan Page, and Michael Tunstall. Side-channel analysis of cryptographic software via early-terminating multiplications. In *ICISC'09*.

[92] Christian Grothoff, Krista Grothoff, Ludmila Alkhutova, Ryan Stutsman, and Mikhail Atallah. Translation-based steganography. In *International Workshop on Information Hiding*, pages 219–233. Springer, 2005.

[93] D. Gullasch, E. Bangerter, and S. Krenn. Cache games – bringing access-based cache attacks on aes to practice. In *S&P'11*.

[94] Marcus Hähnel, Weidong Cui, and Marcus Peinado. High-resolution side channels for untrusted operating systems. In *USENIX ATC'17*.

[95] Juhyeng Han, Seongmin Kim, Taesoo Kim, and Dongsu Han. Toward scaling hardware security module for emerging cloud services. In *Proceedings of the 4th Workshop on System Software for Trusted Execution*, pages 1–6, 2019.

[96] Harveyslash. harveyslash/deep-steganography. https://github.com/harveyslash/Deep-Steganography, Apr 2018.

[97] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, 12 1997.

[98] Nicholas J Hopper. Toward a theory of steganography. Technical report, Carnegie-Mellon University School of Computer Science, 2004.

[99] Nicholas J. Hopper, John Langford, and Luis von Ahn. Provably secure steganography. In Moti Yung, editor, *Advances in Cryptology – CRYPTO 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 77–92, Santa Barbara, CA, USA, August 18–22, 2002. Springer, Heidelberg, Germany.

[100] Thibaut Horel, Sunoo Park, Silas Richelson, and Vinod Vaikuntanathan. How to subvert backdoored encryption: Security against adversaries that decrypt all ciphertexts. In Avrim Blum, editor, *ITCS 2019: 10th Innovations in Theoretical Computer Science Conference*, volume 124, pages 42:1–42:20, San Diego, CA, USA, January 10–12, 2019. LIPIcs.

[101] Amir Houmansadr, Thomas J. Riedl, Nikita Borisov, and Andrew C. Singer. I want my voice to be heard: IP over voice-over-IP for unobservable censorship circumvention. In *ISOC Network and Distributed System Security Symposium – NDSS 2013*, San Diego, CA, USA, February 24–27, 2013. The Internet Society.

[102] D. Hu, L. Wang, W. Jiang, S. Zheng, and B. Li. A novel image steganography method via deep convolutional generative adversarial networks. *IEEE Access*, 6:38303–38314, 2018.

[103] SHIH-YU HUANG and Ping-Sheng Huang. A homophone-based chinese text steganography scheme for chatting applications. *Journal of Information Science & Engineering*, 35(4), 2019.

[104] HuggingFace. huggingface/swift-coreml-transformers.

https://github.com/huggingface/swift-coreml-transformers, Oct 2019.

[105] William Hupp, Adarsh Hasandka, Ricardo Siqueira de Carvalho, and Danish Saleem. Module-ot: a hardware security module for operational technology. In *2020 IEEE Texas Power and Energy Conference (TPEC)*, pages 1–6. IEEE, 2020.

[106] Marcus Hutter. The human knowledge compression contest. http://prize.hutter1.net/, 2006.

[107] SANS institution. Global information assurance certification paper. https://www.giac.org/paper/gsec/1508/overview-hardware-security-modules/102811, 2002.

[108] Intel®. Intel® software guard extensions programming reference, 2014.

[109] Intel®. Intel® 64 and ia-32 architectures software developer's manual volume 3b: System programming guide. 2020.

[110] Intel®. Intel® software guard extensions sdk for linux os developer reference. 2020.

[111] Mika Juuti, Sebastian Szyller, Samuel Marchal, and N Asokan. Prada: protecting against dnn model stealing attacks. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 512–527. IEEE, 2019.

[112] Andrej Karpathy. The unreasonable effectiveness of recurrent neural networks. https://karpathy.github.io/2015/05/21/rnn-effectiveness/, May 2015.

[113] Kohei Kasamatsu, Takahiro Matsuda, Keita Emura, Nuttapong Attrapadung, Goichiro Hanaoka, and Hideki Imai. Time-specific encryption from forward-secure encryption. In *International Conference on Security and Cryptography for Networks*, pages 184–204. Springer, 2012.

[114] Anand Kashyap. The next generation: Hsm approach delivers unparalleld cost/benefit for organizations. https://securitytoday.com/Articles/2018/12/01/The-Next-Generation.aspx?Page=1, Dec 2018.

[115] Marcel Keller. MP-SPDZ: A versatile framework for multi-party computation. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020: 27th Conference on Computer and Communications Security*, pages 1575–1590, Virtual Event, USA, November 9–13, 2020. ACM Press.

[116] Marcel Keller, Emmanuela Orsini, Dragos Rotaru, Peter Scholl, Eduardo Soria-Vazquez, and Srinivas Vivek. Faster secure multi-party computation of AES and DES using lookup tables. In Dieter Gollmann, Atsuko Miyaji, and Hiroaki Kikuchi, editors, *ACNS 17: 15th International Conference on Applied Cryptography and Network Security*, volume 10355 of *Lecture Notes in Computer Science*, pages 229–249, Kanazawa, Japan, July 10–12, 2017. Springer, Heidelberg, Germany.

[117] Yoon Kim, Yacine Jernite, David Sontag, and Alexander M. Rush. Character-aware neural language models. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, pages 2741–2749. AAAI Press, 2016.

[118] Adam Kind. Talk to transformer. https://app.inferkit.com/demo.

[119] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy*, pages 1–19, San Francisco, CA, USA, May 19–23, 2019. IEEE Computer Society Press.

[120] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *CRYPTO'99*.

[121] J. Kodovsky, J. Fridrich, and V. Holub. Ensemble classifiers for steganalysis of digital media. *IEEE Transactions on Information Forensics and Security*, 7(2):432–444, April 2012.

[122] Jing Yu Koh. https://modelzoo.co/.

[123] Hugo Krawczyk. Cryptographic extraction and key derivation: The HKDF scheme.

In Tal Rabin, editor, *Advances in Cryptology – CRYPTO 2010*, volume 6223 of *Lecture Notes in Computer Science*, pages 631–648, Santa Barbara, CA, USA, August 15–19, 2010. Springer, Heidelberg, Germany.

[124] Hugo Krawczyk, Mihir Bellare, and Ran Canetti. HMAC: Keyed-hashing for message authentication. IETF Internet Request for Comments 2104, February 1997.

[125] 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 Nadia Heninger and Patrick Traynor, editors, *USENIX Security 2019: 28th USENIX Security Symposium*, pages 1519–1536, Santa Clara, CA, USA, August 14–16, 2019. USENIX Association.

[126] Tri Van Le. Efficient provably secure public key steganography. Cryptology ePrint Archive, Report 2003/156, 2003. https://eprint.iacr.org/2003/156.

[127] Tri Van Le and Kaoru Kurosawa. Efficient public key steganography secure against adaptively chosen stegotext attacks. Cryptology ePrint Archive, Report 2003/244, 2003. https://eprint.iacr.org/2003/244.

[128] Chang Liu, Austin Harris, Martin Maas, Michael Hicks, Mohit Tiwari, and Elaine Shi. Ghostrider: A hardware-software system for memory trace oblivious computation. *SIGPLAN Not.*, 50(4):87–101, March 2015.

[129] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. Oblivm: A programming framework for secure computation. In *S&P'15*.

[130] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-level cache side-channel attacks are practical. In *S&P'15*.

[131] Daniel Luchaup, Kevin P. Dyer, Somesh Jha, Thomas Ristenpart, and Thomas Shrimpton. Libfte: A toolkit for constructing practical, format-abiding encryption schemes. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 877–891, San Diego, CA, 2014. USENIX Association.

[132] Anna Lysyanskaya and Mira Meyerovich. Provably secure steganography with imperfect sampling. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *PKC 2006: 9th International Conference on Theory and Practice of Public Key Cryptography*, volume 3958 of *Lecture Notes in Computer Science*, pages 123–139, New York, NY, USA, April 24–26, 2006. Springer, Heidelberg, Germany.

[133] Dawn MacKeen. Can new technology make home dialysis a more realistic option? *The New York Times*, Nov 2022. Accessed 2023-02-27.

[134] Bill Marczak, Nicholas Weaver, Jakub Dalek, Roya Ensafi, David Fifield, Sarah McKune, Arn Rey, John Scott-Railton, Ron Deibert, and Vern Paxson. An analysis of china's "great cannon". In *5th {USENIX} Workshop on Free and Open Communications on the Internet ({FOCI} 15)*, 2015.

[135] Matt Burns. Nest thermostat teardown reveals beautiful innards, powerful arm cpu, zigbee radio. https://techcrunch.com/2011/12/22/nest-arm-zigbee/, Dec 2011. Accessed 2/27/2023.

[136] Andrew Mayne. Ai|writer. https://www.aiwriter.app/.

[137] John McCarthy. Recursive functions of symbolic expressions and their computation by machine. 1959.

[138] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative instructions and software model for isolated execution. *HASP'13*.

[139] P. Meng, L. Huang, Z. Chen, W. Yang, and D. Li. Linguistic steganography detection based on perplexity. In *2008 International Conference on MultiMedia and Information Technology*, pages 217–220, Dec 2008.

[140] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. Oblix: An efficient oblivious search index. In *2018 IEEE Symposium on Security and Privacy*, pages 279–296, San Francisco, CA, USA, May 21–23, 2018. IEEE

Computer Society Press.

[141] Thomas Mittelholzer. An information-theoretic approach to steganography and watermarking. In *International Workshop on Information Hiding*, pages 1–16. Springer, 1999.

[142] Hooman Mohajeri Moghaddam, Baiyu Li, Mohammad Derakhshani, and Ian Goldberg. SkypeMorph: protocol obfuscation for Tor bridges. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 2012: 19th Conference on Computer and Communications Security*, pages 97–108, Raleigh, NC, USA, October 16–18, 2012. ACM Press.

[143] Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. Memjam: A false dependency attack against constant-time crypto implementations. *CoRR'17*.

[144] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. Cachezoom: How SGX amplifies the power of cache attacks. *CoRR'17*.

[145] David Molnar, Matt Piotrowski, David Schultz, and David Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In Dongho Won and Seungjoo Kim, editors, *ICISC 05: 8th International Conference on Information Security and Cryptology*, volume 3935 of *Lecture Notes in Computer Science*, pages 156–168, Seoul, Korea, December 1–2, 2006. Springer, Heidelberg, Germany.

[146] Mordor Intelligence. Global smart homes market—growth, analysis, forecast to 2022. https://www.mordorintelligence.com/industry-reports/global-smart-homes-market-industry, 2022.

[147] David M'Raihi, Mihir Bellare, Frank Hoornaert, David Naccache, and Ohad Ranen. RFC 4226: HOTP: An hmac-based one-time password algorithm, 2005.

[148] David M'Raihi, Salah Machani, Mingliang Pei, and Johan Rydell. RFC 6238: TOTP: Time-based one-time password algorithm, 2011.

[149] Moni Naor, Benny Pinkas, and Omer Reingold. Distributed pseudo-random functions and KDCs. In Jacques Stern, editor, *Advances in Cryptology – EUROCRYPT'99*, volume 1592 of *Lecture Notes in Computer Science*, pages 327–346, Prague, Czech Republic, May 2–6, 1999. Springer, Heidelberg, Germany.

[150] K. Nayak, X. S. Wang, S. Ioannidis, U. Weinsberg, N. Taft, and E. Shi. Graphsc: Parallel secure computation made easy. In *S&P'15*.

[151] Alireza Nazari, Nader Sehatbakhsh, Monjur Alam, Alenka Zajic, and Milos Prvulovic. EDDIE: EM-Based Detection of Deviations in Program Execution. In *ISCA'17*.

[152] Masatoshi Nei. F-statistics and analysis of gene diversity in subdivided populations. *Annals of human genetics*, 41(2):225–233, 1977.

[153] NIST. Lightweight cryptography standardization process: Nist selects ascon. https://csrc.nist.gov/News/2023/lightweight-cryptography-nist-selects-ascon, Feb 2023. Accessed 2/27/2023.

[154] Jonathan Oakley, Lu Yu, Xingsi Zhong, Ganesh Kumar Venayagamoorthy, and Richard Brooks. Protocol proxy: An fte-based covert channel. *Computers & Security*, 92:101777, May 2020.

[155] Seong Joon Oh, Bernt Schiele, and Mario Fritz. Towards reverse-engineering black-box neural networks. In *Explainable AI: Interpreting, Explaining and Visualizing Deep Learning*, pages 121–144. Springer, 2019.

[156] Olga Ohrimenko, Felix Schuster, Cedric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. Oblivious multi-party machine learning on trusted processors. In *USENIX Security'16*.

[157] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In *CT-RSA'06*.

[158] Kim Parker, Juliana Menasce Horowitz, and Rachel Minkin. How the coronavirus outbreak has - and hasn't - changed the way americans work. *Pew Research Center*, Dec 2020. Accessed 2023-02-27.

[159] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[160] Kenneth G Paterson and Elizabeth A Quaglia. Time-specific encryption. In *International Conference on Security and Cryptography for Networks*, pages 1–16. Springer, 2010.

[161] Chris Peikert and Brent Waters. Lossy trapdoor functions and their applications. In Richard E. Ladner and Cynthia Dwork, editors, *40th Annual ACM Symposium on Theory of Computing*, pages 187–196, Victoria, BC, Canada, May 17–20, 2008. ACM Press.

[162] Trevor Perrin and Moxie Marlinspike. The double ratchet algorithm. Available at https://whispersystems.org/docs/specifications/doubleratchet/.

[163] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM addressing for cross-cpu attacks. In *USENIX Security'16*.

[164] Pew Research Center. Mobile fact sheet. https://www.pewresearch.org/internet/fact-sheet/mobile/, Apr 2021. Accessed 7/29/2022.

[165] Tran Viet Xuan Phuong, Willy Susilo, Guomin Yang, Jun Yan, and Dongxi Liu. Location based encryption. In Julian Jang-Jaccard and Fuchun Guo, editors, *ACISP 19: 24th Australasian Conference on Information Security and Privacy*, volume 11547 of *Lecture Notes in Computer Science*, pages 21–38, Christchurch, New Zealand, July 3–5, 2019. Springer, Heidelberg, Germany.

[166] Di Qiu, Sherman Lo, Per Enge, Dan Boneh, and Ben Peterson. Geoencryption using loran. In *Proceedings of the 2007 National Technical Meeting of The Institute of Navigation*, pages 104–115, 2007.

[167] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2019.

[168] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI Blog*, 1(8), 2019.

[169] Ram Sundara Raman, Adrian Stoll, Jakub Dalek, Reethika Ramesh, Will Scott, and Roya Ensafi. Measuring the deployment of network censorship filters at global scale. In *Network and Distributed Systems Security (NDSS) Symposium 2020*, 2020.

[170] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In Jaeyeon Jung and Thorsten Holz, editors, *USENIX Security 2015: 24th USENIX Security Symposium*, pages 431–446, Washington, DC, USA, August 12–14, 2015. USENIX Association.

[171] M. G. Reed, P. F. Syverson, and D. M. Goldschlag. Anonymous connections and onion routing. *IEEE Journal on Selected Areas in Communications*, 16(4):482–494, May 1998.

[172] Leonid Reyzin and Scott Russell. Simple stateless steganography. Cryptology ePrint Archive, Report 2003/093, 2003. https://eprint.iacr.org/2003/093.

[173] Sean Robertson. spro/char-rnn.pytorch. https://github.com/spro/char-rnn.pytorch, Dec 2017.

[174] Robinhood. Two-factor authentication. https://robinhood.com/us/en/support/articles/twofactor-authentication/, 2022. Accessed 2022-07-29.

[175] Bryan Robinson. Remote Work is Here to Stay And Will Increase Into 2023, Experts Say. *Forbes*, Feb 2022.

[176] Dan Rockmore. What happens when machines learn to write poetry. https://www.newyorker.com/culture/annals-of-inquiry/the-mechanical-muse, Jan 2020.

[177] Tim Ruffing, Jonas Schneider, and Aniket Kate. Identity-based steganography and its applications to censorship resistance. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013: 20th Conference on Computer and Communications Security*, pages 1461–1464, Berlin, Germany, November 4–8, 2013. ACM Press.

[178] Sylvain Ruhault. SoK: Security models for pseudo-random number generators. *IACR Transactions on Symmetric Cryptology*, 2017(1):506–544, 2017.

[179] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, Jan 2003.

[180] Md Nazmus Sadat, Md Momin Al Aziz, Noman Mohammed, Feng Chen, Shuang Wang, and Xiaoqian Jiang. SAFETY: Secure GWAS in federated environment through a hybrid solution with Intel SGX and homomorphic encryption. *arXiv preprint arXiv:1703.02577*, 2017.

[181] Ahmed Salem, Yang Zhang, Mathias Humbert, Pascal Berrang, Mario Fritz, and Michael Backes. Ml-leaks: Model and data independent membership inference attacks and defenses on machine learning models. *arXiv preprint arXiv:1806.01246*, 2018.

[182] A. Sarkar, K. Solanki, and B. S. Manjunath. Secure steganography: Statistical restoration in the transform domain with best integer perturbations to pixel values. In *IEEE International Conference on Image Processing (ICIP)*, Sep 2007.

[183] Sajin Sasy, Sergey Gorbunov, and Christopher W. Fletcher. Zerotrace : Oblivious memory primitives from intel sgx. In *NDSS'18*.

[184] Sajin Sasy, Sergey Gorbunov, and Christopher W. Fletcher. ZeroTrace : Oblivious

memory primitives from intel SGX. In *ISOC Network and Distributed System Security Symposium – NDSS 2018*, San Diego, CA, USA, February 18–21, 2018. The Internet Society.

[185] Sarah Scheffler and Mayank Varia. Protecting cryptography against compelled self-incrimination. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021: 30th USENIX Security Symposium*, pages 591–608. USENIX Association, August 11–13, 2021.

[186] Logan Scott and Dorothy E Denning. A location based encryption technique and some of its applications. In *Proceedings of the 2003 National Technical Meeting of The Institute of Navigation*, pages 734–740, 2003.

[187] Ayon Sen, Scott Alfeld, Xuezhou Zhang, Ara Vartanian, Yuzhe Ma, and Xiaojin Zhu. Training set camouflage. *Decision and Game Theory for Security*, page 59–79, 2018.

[188] Jayasree Sengupta, Sushmita Ruj, and Sipra Das Bit. A comprehensive survey on attacks, security issues and blockchain solutions for iot and iiot. *Journal of Network and Computer Applications*, 149:102481, 2020.

[189] Adrian Shahbaz. Freedom on the net 2018. Available at https://freedomhouse.org/report/freedom-net/freedom-net-2018/rise-digital-authoritarianism, 2018.

[190] Fahad Shaon, Murat Kantarcioglu, Zhiqiang Lin, and Latifur Khan. SGX-BigMatrix: A practical encrypted data analytic framework with trusted processors. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017: 24th Conference on Computer and Communications Security*, pages 1211–1228, Dallas, TX, USA, October 31 – November 2, 2017. ACM Press.

[191] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. Panoply: Low-TCB linux applications with sgx enclaves. In *NDSS'17*.

[192] Mohammad Shirali-Shahreza and M. H. Shirali-Shahreza. Text steganography in

sms. *2007 International Conference on Convergence Information Technology (ICCIT 2007)*, pages 2260–2265, 2007.

[193] Prakash Shrestha and Nitesh Saxena. Listening watch: Wearable two-factor authentication using speech signals resilient to near-far attacks. In *Proceedings of the 11th ACM conference on security & privacy in wireless and mobile networks*, pages 99–110, 2018.

[194] Gustavus J. Simmons. The prisoners' problem and the subliminal channel. In David Chaum, editor, *Advances in Cryptology – CRYPTO'83*, pages 51–67, Santa Barbara, CA, USA, 1983. Plenum Press, New York, USA.

[195] Tom Simonite. Apple's latest iphones are packed with ai smarts. https://www.wired.com/story/apples-latest-iphones-packed-with-ai-smarts/.

[196] D. Skarlatos, M. Yan, B. Gopireddy, R. Sprabery, J. Torrellas, and C. W. Fletcher. Microscope: Enabling microarchitectural replay attacks. In *ISCA'19*.

[197] K. Solanki, K. Sullivan, U. Madhow, B. S. Manjunath, and S. Chandrasekaran. Provably secure steganography: Achieving zero k-l divergence using statistical restoration. In *2006 International Conference on Image Processing*, pages 125–128, Oct 2006.

[198] E. M. Songhori, S. U. Hussain, A. Sadeghi, T. Schneider, and F. Koushanfar. Tinygarble: Highly compressed and scalable sequential garbled circuits. In *S&P'15*.

[199] Frank Stajano. Pico: No more passwords! In *Security Protocols XIX: 19th International Workshop, Cambridge, UK, March 28-30, 2011, Revised Selected Papers 19*, pages 49–81. Springer, 2011.

[200] Statista. Digital market - smart home. https://www.statista.com/outlook/dmo/smart-home/worldwide, 2022.

[201] Emil Stefanov, Marten van Dijk, Elaine Shi, T-H. Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path oram: An extremely simple

oblivious ram protocol. *CCS'13*.

[202] Pramod Subramanyan, Rohit Sinha, Ilia Lebedev, Srinivas Devadas, and Sanjit A. Seshia. A formal foundation for secure remote execution of enclaves. In *CCS'17*.

[203] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure Program Execution via Dynamic Information Flow Tracking. In *ASPLOS'04*.

[204] Kenneth Sullivan, Kaushal Solanki, B. S. Manjunath, Upamanyu Madhow, and Shivkumar Chandrasekaran. Determining achievable rates for secure, zero divergence, steganography. In *ICIP*, pages 121–124. IEEE, 2006.

[205] Kun Tang, Kevin R Thornton, and Mark Stoneking. A new approach for using genome scans to detect recent positive selection in the human genome. *PLoS biology*, 5(7):e171, 2007.

[206] TechInfoDepot. Belkin wemo light switch (f7c030). http://en.techinfodepot.shoutwiki.com/wiki/Belkin_WeMo_Light_Switch_(F7C030), Jan 2023. Accessed 2/27/2023.

[207] Shruti Tople and Prateek Saxena. On the trade-offs in oblivious execution techniques. In Michalis Polychronakis and Michael Meier, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer'17.

[208] Tor Project. The tor project: Privacy and freedom online. https://www.torproject.org/.

[209] M. C. Tschantz, S. Afroz, Anonymous, and V. Paxson. Sok: Towards grounding censorship circumvention in empiricism. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 914–933, May 2016.

[210] Nirvan Tyagi, Muhammad Haris Mughees, Thomas Ristenpart, and Ian Miers. BurnBox: Self-revocable encryption in a world of compelled access. In William Enck and Adrienne Porter Felt, editors, *USENIX Security 2018: 27th USENIX Security Symposium*, pages 445–461, Baltimore, MD, USA, August 15–17, 2018. USENIX

Association.

[211] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control. In *SysTEX'17*.

[212] Arjen van Dalen. The algorithms behind the headlines. *Journalism Practice*, 6(5-6):648–658, 2012.

[213] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, undefinedukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 6000–6010, Red Hook, NY, USA, 2017. Curran Associates Inc.

[214] Denis Volkhonskiy, Ivan Nazarov, Boris Borisenko, and Evgeny Burnaev. Steganographic generative adversarial networks, 2017.

[215] Luis von Ahn and Nicholas J. Hopper. Public-key steganography. In Christian Cachin and Jan Camenisch, editors, *Advances in Cryptology – EUROCRYPT 2004*, volume 3027 of *Lecture Notes in Computer Science*, pages 323–341, Interlaken, Switzerland, May 2–6, 2004. Springer, Heidelberg, Germany.

[216] Qiyan Wang, Xun Gong, Giang T. K. Nguyen, Amir Houmansadr, and Nikita Borisov. CensorSpoofer: asymmetric communication using IP spoofing for censorship-resistant web browsing. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 2012: 19th Conference on Computer and Communications Security*, pages 121–132, Raleigh, NC, USA, October 16–18, 2012. ACM Press.

[217] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. In *CCS'17*.

[218] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Global-scale secure multiparty computation. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017: 24th Conference on Computer and Communications Security*,

pages 39–56, Dallas, TX, USA, October 31 – November 2, 2017. ACM Press.

[219] Xiao Shaun Wang, Kartik Nayak, Chang Liu, T-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious data structures. *IACR'14*.

[220] Yash Wate. How to Enable Two-Factor Authentication on Facebook, Instagram, and Twitter. https://techpp.com/2020/02/03/enable-two-factor-authentication-instagram-facebook-twitter/, Nov 2021. Accessed 2022-07-29.

[221] Zachary Weinberg, Jeffrey Wang, Vinod Yegneswaran, Linda Briesemeister, Steven Cheung, Frank Wang, and Dan Boneh. StegoTorus: a camouflage proxy for the Tor anonymity system. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 2012: 19th Conference on Computer and Communications Security*, pages 109–120, Raleigh, NC, USA, October 16–18, 2012. ACM Press.

[222] Bruce S Weir and C Clark Cockerham. Estimating F-statistics for the analysis of population structure. *evolution*, 38(6):1358–1370, 1984.

[223] WhatsApp. WhatsApp Encryption Overview. Available at https://scontent.whatsapp.net/v/t61/68135620_760356657751682_6212997528851833559_n.pdf/WhatsApp-Security-Whitepaper.pdf, December 2017.

[224] Alex Wilson, Phil Blunsom, and Andrew Ker. Detection of steganographic techniques on twitter. *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, 2015.

[225] Philipp Winter, Tobias Pulls, and Jürgen Fuß. Scramblesuit: A polymorph network protocol to circumvent censorship. *CoRR*, abs/1305.3199, 2013.

[226] Marko Wolf and Timo Gendrullis. Design, implementation, and evaluation of a vehicular hardware security module. In *International Conference on Information Security and Cryptology*, pages 302–318. Springer, 2011.

[227] Pin Wu, Yang Yang, and Xiaoqiang Li. Stegnet: Mega image steganography capacity with deep convolutional network. *Future Internet*, 10(6):54, Jun 2018.

[228] Eric Wustrow, Colleen M Swanson, and J Alex Halderman. Tapdance: End-to-middle anticensorship without flow blocking. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 159–174, 2014.

[229] Eric Wustrow, Scott Wolchok, Ian Goldberg, and J. Alex Halderman. Telex: Anticensorship in the network infrastructure. In *Proceedings of the 20th USENIX Security Symposium*, August 2011.

[230] Lingyun Xiang. Reversible natural language watermarking using synonym substitution and arithmetic coding, 2018.

[231] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *S&P'15*.

[232] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. Attack Directories, Not Caches: Side Channel Attacks in a Non-Inclusive World. In *IEEE S&P*, 2019.

[233] K. Yang, M. Hicks, Q. Dong, T. Austin, and D. Sylvester. A2: Analog Malicious Hardware. In *S&P'16*.

[234] Z. Yang, X. Guo, Z. Chen, Y. Huang, and Y. Zhang. Rnn-stega: Linguistic steganography based on recurrent neural networks. *IEEE Transactions on Information Forensics and Security*, 14(5):1280–1295, May 2019.

[235] Zhongliang Yang, Yongfeng Huang, and Yu-Jin Zhang. A fast and efficient text steganalysis method. *IEEE Signal Processing Letters*, 26:627–631, 2019.

[236] Zhongliang Yang, Shuyu Jin, Yongfeng Huang, Yujin Zhang, and Hui Li. Automatically generate steganographic text based on markov model and huffman coding, 2018.

[237] Zhongliang Yang, Ke Wang, Jian Li, Yongfeng Huang, and Yujin Zhang. Ts-rnn: Text steganalysis based on recurrent neural networks. *IEEE Signal Processing Letters*, page 1–1, 2019.

[238] Zhongliang Yang, Nan Wei, Junyi Sheng, Yongfeng Huang, and Yu-Jin Zhang. Ts-cnn: Text steganalysis from semantic space based on convolutional neural network, 2018.

[239] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th Annual Symposium on Foundations of Computer Science*, pages 162–167, Toronto, Ontario, Canada, October 27–29, 1986. IEEE Computer Society Press.

[240] Yuval Yarom and Katrina Falkner. Flush+Reload: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Security'14*.

[241] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: a timing attack on OpenSSL constant-time RSA. volume 7, pages 99–112. Springer, Heidelberg, Germany, June 2017.

[242] Jiyong Yu, Lucas Hsiung, Mohamad El Hajj, and Christopher W. Fletcher. Data oblivious ISA extensions for side channel-resistant and high performance computing. In *ISOC Network and Distributed System Security Symposium – NDSS 2019*, San Diego, CA, USA, February 24–27, 2019. The Internet Society.

[243] Zhenshan Yu, Liusheng Huang, Zhili Chen, Lingjun Li, Xinxin Zhao, and Youwen Zhu. Steganalysis of synonym-substitution based natural language watermarking, 2009.

[244] C. Bormann Z. Shelby, K. Hartke. The constrained application protocol (coap). RFC 7252, RFC Editor, Jun 2014.

[245] Samee Zahur and David Evans. Circuit structures for improving efficiency of security and privacy tools. In *S&P'13*.

[246] Samee Zahur and David Evans. Obliv-c: A language for extensible data-oblivious

computation. *IACR'15*.

[247] Jiansong Zhang, Zeyu Wang, Zhice Yang, and Qian Zhang. Proximity based iot device authentication. In *IEEE INFOCOM 2017-IEEE conference on computer communications*, pages 1–9. IEEE, 2017.

[248] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *NSDI'17*.

[249] Yaoming Zhu, Sidi Lu, Lei Zheng, Jiaxian Guo, Weinan Zhang, Jun Wang, and Yong Yu. Texygen: A benchmarking platform for text generation models. In *The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval*, pages 1097–1100. ACM, 2018.

[250] Zachary M. Ziegler, Yuntian Deng, and Alexander M. Rush. Neural linguistic steganography, 2019.

[251] Maximilian Zinkus, Tushar M. Jois, and Matthew Green. SoK: Cryptographic confidentiality of data on mobile devices. *Proceedings on Privacy Enhancing Technologies*, 2022(1):586–607, January 2022.

[252] Jan Zöllner, Hannes Federrath, Herbert Klimant, Andreas Pfitzmann, Rudi Piotraschke, Andreas Westfeld, Guntram Wicke, and Gritta Wolf. Modeling the security of steganographic systems. In *International Workshop on Information Hiding*, pages 344–354. Springer, 1998.