DATA CONFIDENTIALITY FOR ALL: NEW METHODS IN ATTACK AND DEFENSE

by

Maximilian A. Zinkus

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

Baltimore, Maryland
September, 2024

Abstract

For the past half-century since the advent of modern cryptography, perhaps most distinctly demarcated by Diffie and Hellman in 1976, dramatic advances in cryptographic theory and practice have propelled us from a world where secrets were best hidden by obfuscation and hope to one where private, authenticated communication is the almost invisible backdrop of all electronic communication and where advanced cryptosystems secure previously unimaginable functionalities such as privacy-preserving computation and even novel digital financial systems.

One connecting thread in this history is that cryptographic constructions and cryptanalysis have largely been done by, and for, cryptographers. This is perhaps most clearly evidenced in the folklore maxim that almost every computer scientist hears at some point: "don't roll your own crypto." Recent efforts to promote usability and transparency in cryptography are poignantly summarized in seminal critiques of the complexity and opaqueness of cryptographic tools.

The lack of accessibility in both building and evaluating cryptosystems is far worse than just an inconvenience of consulting experts. As Rogaway famously described, cryptography is a mechanism of power – the power to control the flow of information, to protect data sensitive to us, and to interact with modern technology systems. Thus, accessibility is a matter of equity as much as it is one of efficiency. When viewed through this lens, the lack of accessibility to cryptography expertise can be clearly understood as a problem at the intersection of the societal and the technical.

Throughout this work, we explore opportunities to leverage automation, familiar tools, and commodity platforms to develop novel cryptographic methods. In doing so, we seek to democratize access to cryptography knowledge, implementation, and evaluation. Technology alone cannot be expected to solve societal problems. However, only by including considerations of accessibility and usability into our development of novel methods can we hope to pursue efforts which serve to mitigate the impacts of these societal challenges.

Primary Reader and Advisor: Matthew D. Green

Secondary Readers: Yinzhi Cao, Michael Rushanan

Acknowledgments

The road to completing a doctoral degree is fraught with perils; rarely, however, are they the ones we expected when we start. I want to thank everyone who has supported me throughout this journey, principally my partner Sophia, my parents Alfa and Maria, and my advisor, Dr. Matthew Green. Thank you for enduring with me through fits and starts of productivity, through hard days rewriting and resubmitting papers, through the labyrinths of academic administrivia, and through a global pandemic. I would not have completed this without you, and so the success is ours to share.

Thank you as well to many mentors and friends I've made along the way, the instructors and peers whose insight helped me develop as a researcher, as an engineer, and most importantly, as a person. I want to particularly thank my professors at Cal Poly San Luis Obispo and Johns Hopkins University, and everyone else who shared their experience, expertise, and excitement with me when our paths intersected. You have all rigorously proven the adage that it is the journey itself which truly matters.

Finally, thank you to the handful of students I met who tirelessly strove to advance academia in terms of its diversity and inclusivity, especially those who persisted when I, after whatever small ways I was able to contribute, could not. Your efforts are the ripples which collect and magnify to slowly turn the tide. The status quo, no matter how deeply entrenched it may be today, will bend under the weight of the passion and energy you bring to the cause.

Contents

Abstract **ii**

Acknowledgments **iv**

Contents **v**

List of Tables **xi**

List of Figures **xii**

1 Introduction **1**

2 Delphinium **11**

2.1 Intuition 16

2.2 Preliminaries 27

2.2.1 Encryption Schemes and Malleability 27

2.2.1.1 Malleation Functions 28

2.2.2 Theory Solvers and Model Counting 31

2.2.3 Format Checking Functions 33

2.3 Constructions 34

2.3.1 Main Algorithm 34
2.3.2 Realizing the Max#SAT Oracle 36
2.3.3 Additional Algorithms 37
2.4 Proof that `DeriveAttack` is Greedy 39
2.4.1 Preliminaries 40
2.4.2 Proof 42
2.4.2.1 Base Case 42
2.4.2.2 Strong Induction 43
2.4.2.3 Max#SAT Aborts 44
2.5 The Greedy Algorithm can be Optimal 44
2.5.1 Preliminaries 45
2.5.2 Observations 46
2.5.3 The Optimal Attack 47
2.5.4 The Greedy Attack 48
2.5.5 Discussion 49
2.6 The Greedy Algorithm can be Suboptimal 49
2.6.1 Preliminaries 50
2.6.1.1 Parity Check 51
2.6.1.2 Padding Check 51
2.6.2 Observations 52
2.6.3 The Optimal Attack 52
2.6.4 The Greedy Attack 54
2.6.5 Discussion 55

2.6.5.1 Edge Cases 55
2.7 Realizing Plaintext Weight Functions 57
2.8 Prototype Implementation 58
2.8.1 Architecture Overview 58
2.8.2 Implementation Details 59
2.8.2.1 Implementing Malleation Functions 61
2.8.3 Further Implementation Considerations 64
2.8.3.1 Selecting SAT and SMT solvers 64
2.8.3.2 Employing parallelization 66
2.8.3.3 Tseitin conversion and CNFXOR 67
2.8.4 Software 67
2.8.5 Extensions 68
2.9 Format Check Implementations 68
2.9.1 PKCS #7 padding 69
2.9.2 AWS Session Ticket 69
2.9.3 Bitwise padding 70
2.10 Experiments 70
2.10.1 Experimental Setup 71
2.10.2 Experiments with Stream Ciphers 72
2.10.3 Ciphers with Truncation 75
2.10.4 CBC mode 79
2.11 Additional Experimental Results 81
2.11.1 PKCS #7 81

2.11.2 Bitwise Padding Format Tests Over t 82

2.11.3 Differences between Short and Long Exclusive-OR 84

2.11.4 CRC 86

2.11.5 Sudoku 86

2.12 Discussion 88

2.12.1 Related Work 88

2.12.2 Open Problems 89

3 McFIL **91**

3.1 Intuition: Maximizing Leakage 95

3.1.1 A Minimal Example: Binary AND 96

3.1.2 A Further Example: Yao's Millionaires 97

3.1.3 Overview of McFIL 99

3.2 Technical Background 104

3.2.1 SAT and SMT 104

3.2.2 Extensions to SAT 105

3.2.3 Prior Work 107

3.3 McFIL 108

3.3.1 Beyond Predicate Functions 110

3.3.2 Implementation 115

3.4 Evaluation 117

3.4.1 Selected Functionalities 117

3.4.1.1 Millionaires Problem 118

3.4.1.2 Dual Execution 119

3.4.1.3 Danish Sugar Beets Auction . . . 120
3.4.1.4 BWWC . . . 121
3.5 A Note on Zero-Knowledge Range Proofs . . . 122
3.6 Solver Domain-Specific Language Examples . . . 123
3.6.1 Evaluation Results . . . 125
3.6.1.1 Millionaires Problem . . . 125
3.6.1.2 Dual Execution . . . 126
3.6.1.3 Danish Sugar Beets Auction . . . 126
3.6.1.4 BWWC . . . 127
3.7 Additional Evaluation . . . 130
3.7.1 Contributed Benchmarks . . . 132
3.8 Discussion . . . 134
3.8.1 Practicality . . . 134
3.8.2 Open Problems . . . 135

4 SocIoTy **138**

4.1 SocIoTy . . . 142
4.2 Background . . . 145
4.3 Threat Model . . . 149
4.4 At-Home Cryptography . . . 154
4.4.1 At-Home Cryptographic Applications . . . 155
4.4.2 At-Home Cryptographic Primitives. . . . 156
4.5 SocIoTy . . . 158
4.5.1 At-Home Authentication . . . 160

4.5.2 At-Home Encryption 163
4.5.3 Handling Device Churn 166
4.5.4 Security Intuition 171
4.6 Evaluation 172
4.6.1 Implementation 172
4.6.2 Experimental Setup 174
4.6.3 Results 175
4.7 Summary Statistics 176
4.7.1 Discussion. 176
4.8 In-Depth Security Analysis 178
4.9 Related Work 181

5 Conclusion **184**

Bibliography **186**

List of Tables

3.1 McFIL Evaluated Functionalities 118

3.2 CNF Size in Clauses over Target Bits 119

3.3 Evaluation Results 120

3.4 Additional Sugar Beets Auction Evaluation 132

3.5 CNF Benchmarks Overview 133

4.1 Summary statistics for the distributions in Figure 4.5a. 176

4.2 Summary statistics for the distributions in Figure 4.5b. 176

4.3 A comparison of related work with similar goals to those of SocIoTy. . . . 182

List of Figures

2.1 Output of a format oracle attack that our algorithms developed against a bitwise padding check oracle $\mathsf{F}_{\mathsf{bitpad}}$ (see §2.10.2 for a full description). The original ciphertext is a valid 128-bit (random) padded message encrypted using a stream cipher. Each row of the bitmap represents a *malleation string* that was exclusive-ORed with the ciphertext prior to making a decryption query. 13

2.2 Left: illustration of a plaintext candidate space defined by G_{i-1}, highlighting the two subsets $\mathcal{M}_0, \mathcal{M}_1$ induced by a specific malleation string $\mathbf{S}$. Right: the candidate space defined by G_i, in which many candidates have been eliminated following an oracle response $b = 1$. 23

2.3 Architecture of `Delphinium`. 60

2.4 A contiguous set of malleation queries made by `Delphinium` during a simulated CBC attack. The rightmost bits signal truncation (from left or right). 80

2.5 PKCS #7 malleation bitmap with characteristic “staircase” pattern as the attack progresses through the ciphertext right to left. The attack proceeds top to bottom, with one row per iteration of the attack. 83

2.6 Malleation strings for 128-bit bitwise padding ($\mathsf{F}_{\mathsf{bitpad}}$) using a stream cipher. Experiments consider several different values of t, all with $\delta = 0.5$. Each uses a different random 128-bit message, which can account for some variability in the attack progress. For $t = 1$ the total number of queries is too large to show, and so we only present approximately the first 10% of the full attack transcript. 84

2.7 Approximate number of removed candidate messages for each query in the attack, with parity constraints of size $\log n$ (i.e. short). 85

2.8 Approximate number of removed candidate messages for each query in the attack, with parity constraints of size $\frac{n}{2}$ (i.e. long). 85

2.9 Representation of a malleation string an attack on a CRC-8 where the length of the data to compute the CRC over is 3 bytes long. 86

2.10 An example of an attack against the Sudoku format check. When the solver has narrowed a square to some candidate entries, multiple numbers are shown in a given square. Squares which are changed from their original values by a malleation are colored in light red. 87

3.1 In a fully-automated analysis of Yao's Millionaires, McFIL "discovers" binary search. Blue diamonds are adversary inputs, true binary search shown in solid red. 96

3.2 McFIL workflow; || denotes parallelization 100

3.3 CNF of $x1 \oplus x2 = x3$ 104

3.4 Architecture of McFIL 109

3.5 Remaining candidate solutions per query in a single trial of 64-bit Yao's Millionaires 123

3.6 Remaining candidate solutions per query in a single trial of Dual Execution (Affine Predicates) 127

3.7 Remaining candidate solutions (log scale) per query 131

4.1 System architecture of SocIoTy. An authorized device interacts with the smart home to perform at-home cryptographic operations for interaction with remote services. 143

4.2 Overview of the SocIoTy threat model. 149

4.3 SocIoTy for at-home authentication. 161

4.4 SocIoTy for at-home encryption: request, generate, encrypt, send, efface. 166

4.5 Evaluation results on 50 runs of $\mathsf{TOTP}_{\text{home}}$ for in-protocol and out-of-protocol churn on AES256 and MiMC. 171

Chapter 1

Introduction

For the past half-century since the advent of modern cryptography, perhaps most distinctly demarcated by Diffie and Hellman in 1976 [DH76], dramatic advances in cryptographic theory and practice have propelled us from a world where secrets were best hidden by obfuscation [Sma16] and hope [BB02] to one where private, authenticated communication is the almost invisible backdrop of all electronic communication [Tur14] and where advanced cryptosystems secure previously unimaginable functionalities such as privacy-preserving computation [Yao82; Yao86a; BOGW88; Gen09] and even novel digital financial systems [Nak08; But+13].

One connecting thread in this history is that cryptographic constructions and cryptanalysis have largely been done by, and for, cryptographers. This is perhaps most clearly evidenced in the folklore maxim that almost every computer scientist hears at some point: “don’t roll your own crypto.” Recent efforts to promote usability and transparency in cryptography (surveyed in works such as [Swe+09]) are poignantly summarized in seminal critiques of the complexity and opaqueness of cryptographic tools such as the comical yet vital work “Why Johnny Can’t Encrypt” and others that have followed [WT99; Ruo+15].

The lack of accessibility in both building and evaluating cryptosystems is far worse than just an inconvenience of consulting experts. As Rogaway famously described [Rog15], cryptography is a mechanism of power – the power to control the flow of information, to protect data sensitive to us, and to interact with modern technology systems. Thus, accessibility is a matter of equity as much as it is one of efficiency. When viewed through this lens, the lack of accessibility to cryptography expertise can be clearly understood as a problem at the intersection of the societal and the technical.

Throughout this work, we explore opportunities to leverage automation, familiar tools, and commodity platforms to develop novel cryptographic methods. In doing so, we seek to democratize access to cryptography knowledge, implementation, and evaluation. Technology alone cannot be expected to solve societal problems. However, only by including considerations of accessibility and usability into our development of novel methods can we hope to pursue efforts which serve to mitigate the impacts of these societal challenges.

The state of the art. A variety of works exist in the research literature on the intersection of usability, in the form of automation or at least tooling, and security for cryptography and cryptographic systems. We consider a wide range of modern related works throughout these chapters, ranging from concrete instantiations of attacks seek to optimize and automate, to tools that exist largely within the theoretical realm or which incur extensive time- or resource-based costs to be effective. The unifying theme across these works is that even the cutting edge of cryptography discoveries limits its valuation of practical implementations and pragmatic usage trade-offs. In many cases, implementations are described or even directly provided in the form of functioning software. However, these implementations are often designed for expert use, or require at least some relevant domain knowledge in order to operate them successfully or securely. As a result, there is some extant gap, a barrier between what we know we can accomplish with technology and cryptography, and the realities of the systems accessible to end-users.

Our approaches. We approach this problem through a humanistic lens, analyzing impact and context of various advancements in cryptography. Our goal is to provide contributions that empower end users or practitioners without requiring domain expertise in cryptography. By providing expressive domain-specific languages, we empower practitioners to access cryptography through familiar interfaces; using powerful and flexible solving tools such as SAT and SMT, we expand the range of concrete problems which our tools can provide direct answers for; and in the constructive case, we design systems and protocols to operate on common commodity devices such as smartphones and home IoT devices. Whether in uncovering new methods of attack or evaluation of purportedly secure encryption or computation schemes, or in applying those methods and developing new defenses to solve practical challenges facing individual users of cryptographic tools, the underlying goal and resulting character of our contributions is centered on improving the accessibility of modern cryptography and the extent of capabilities available to its users.

Our contributions. In this work we describe a number of practical and theoretical contributions to the research literature, in addition to software implementations of our tools and systems along with rigorous accompanying evaluations of their performance and security. We describe a novel method of discovering chosen-ciphertext attacks [Bel96; NY90; BN00] by instantiating recently-developed methods in SAT solving [FRS17] with novel, enabling heuristics which bridge the gap between theoretical correctness and practical feasibility. We then advance this method, generalizing and dramatically expanding its efficiency and applicability to an expanse of cryptographic analyses, and realize novel methods in quantifying information leakage with the accompanying, practically-motivated capability to concretely generate leakage-driven attacks. Finally, we contribute a novel system and protocol, resilient by design to the attacks we describe in prior chapters, to demonstrate the constructive use of accessible cryptography and "close the loop" between uncovering new attacks and designing new defenses.

Outline of this work. In the remainder of Chapter 1, we provide broad background information in the form of technical preliminaries. In Chapter 2, we describe Delphinium, our approach to automating chosen ciphertext attacks. In Chapter 3, we describe McFIL, our approach to automating generalized information leakage attacks, building off of the foundation of Delphinium, and applying these capabilities to secure multi-party computation, fully homomorphic encryption, and zero-knowledge proofs. Then, in Chapter 4, we provide SocIoTy, a novel framework enabled by secure multi-party computation to provide novel cryptographic capabilities to end-users. Finally, in Chapter 5, we provide summary discussion and review the open problems established throughout the course of this work.

Previous publications. Portions of this work have previously been published in other venues. Chapters 2 and 3 consist of material which appeared in the Proceedings of Usenix Security 2018 and 2023, respectively. Similarly, Chapter 4 consists of work that was originally published in the Proceedings on Privacy Enhancing Technologies 2024. We will provide direct citations at the start of each chapter.

Preliminaries

In this section we provide a number of definitions and descriptions to guide the reader throughout this work. In the spirit of accessibility and usability in cryptography which underpins this work, for the purely cryptographic definitions we elide the rigid and more opaque formalized definitions for more clearly intuitive descriptions.

Symmetric encryption. A symmetric encryption scheme consists of three algorithms, usually `Enc`, `Dec`, and `Keygen`, which respectively encipher (hide) a message given a cryptographic key (derived from secret randomness), return enciphered messages to their original plaintext given the same key, and generate keys from some source (pseudo)randomness. The defining characteristic of symmetric encryption is that the key used to encrypt is the same as the key used to decrypt, and therefore must generally be transmitted securely outside of the scheme, such as with a physical hand-off or key agreement protocols such as Diffie-Hellman key exchange [Mer78].

Asymmetric encryption. Similarly, asymmetric encryption consists of the same three algorithms, but with the defining characteristic that the key used to encrypt (the "public" key) and the key used to decrypt (the "private" key) are different. Indeed, as their names might imply, the public key is generally shared widely, enabling anyone to encrypt messages to a given recipient. The private key, correspondingly, must be kept private as it allows the recipient to decrypt those messages. In some descriptions, `Keygen` may only generate a private key, and some other associated algorithm generates a public key from the private key, either deterministically or with randomization if given (pseudo)random input(s).

Asymmetric authentication. Asymmetric authentication can be understood as a sort of converse to asymmetric encryption. The private key is used to generate a "signature" given a message (and usually some (pseudo)randomness) using `Sign`. This signature, along with the message and the corresponding public key, can be provided to `Verify`, which outputs (in effect) `accept` or `reject` if the signature data was correctly generated using the corresponding private key.

Hash functions. Cryptographic hash functions are algorithms which take arbitrary inputs (e.g. byte strings of any length) and output fixed-length "hashes." These outputs can be used as a kind of "fingerprint" for the input, in that any given input will with high probability output a unique hash. However, as defined, these functions cannot exist; infinite inputs mapping to a finite (even if very large) number of outputs necessitates the existence of inputs which map to the same output (a "collision"). However, hash functions rely on these collisions to be computationally infeasible to find. Cryptographic hash functions satisfy the following related properties: one-way (or "pre-image resistance"), second pre-image resistance, and collision-resistance. The first implies that given a hash function output, a computationally-bound adversary will with vanishing probability discover a corresponding input for that output. The second requires that given an input and its corresponding output, a computationally-bound adversary will with vanishing probability find a second input which has the same hash output. Finally, the third property requires that a computationally-bound adversary will with only vanishing probability find two inputs which hash to the same output.

Chosen plaintext attacks. One class of attacks against encryption schemes is the "chosen plaintext attack." In these attacks, an adversary is able to receive encryptions of any plaintext message they choose (adaptively, in some models, meaning that further plaintexts can be chosen after seeing encryptions of prior messages). Then, to succeed in the attack, the attack must essentially be able to "break" the encryption scheme using the information they have gained. This is formalized as the attacker requesting encryptions of two messages (that were not previously returned as encryptions), and the encryption scheme returning the encryption of one or the other, at random. The adversary "wins" if they are able to identify which of the two messages they have received the ciphertext for. The definition of security against such an attack is that any (computationally-bounded) adversary must have only negligibly more than 50%

chance to guess correctly.

Chosen ciphertext attacks. The second major class of attacks against encryption schemes is the "chosen ciphertext attack." In this class of attacks, an additional capability is given to the attacker: to request *decryptions* of arbitrarily chosen ciphertexts, in addition to encryptions as in chosen plaintext attacks. Then, the adversary provides two new plaintexts, and the encryption scheme must provide a new ciphertext (not previously requested for encryption/decryption by the adversary) which is the encryption of one or the other, at random. In a further *adaptive* model, the adversary may then request another round of interactions with the scheme (aside from the challenge ciphertext). In either case, the adversary must then make a guess as to whether the first or second message was encrypted, and they "win" upon guessing correctly, with the same security definition: no more than negligibly more than 50% "advantage" in guessing correctly.

Satisfiability. Satisfiability is a classic problem in computer science. It relates to evaluating the truthiness of a conglomeration of Boolean operations over some set of Boolean variables (true/false values). These operations are traditional Boolean operators (`and`, `or`, `not`, `xor`, etc.). Commonly, these *formulae* are expressed in *Conjunctive Normal Form* (CNF), which simply means an "`AND` of `OR`s" (where any terms of the `OR`s may be negated with a `NOT`). CNF is sufficiently expressive to capture any Boolean formula [Tse68]. Satisfiability asks, *is there an assignment to the variables of the formula, such that the formula evaluates to true?* It turns out that the complexity of computing the answer in arbitrarily complex formula is in the problem class `NP`, and indeed is `NP-complete`, roughly implying that it can be equated (or "reduced") to a solution to any other problem in `NP` and thus is at least as hard as any of them. Satisfiability is practically interesting for two reasons. First, it is often the most easily manipulated problem in NP due to efficient expression in CNF and the simplicity of Boolean values and operators. Second, any finite computer program can be approximated by a Boolean circuit simply by expressing computational steps as manipulations of bits, and then encoding those bitwise manipulations as Boolean operators in a formula. Thus, solving satisfiability over intentionally-constructed formulae can express results of practical interest, such as inputs which cause a program to reach a certain desired state.

Intrinsic or inherent leakage. In quantitative information theories, "leakage" is defined as a quantification of information which an external party (e.g. an adversary, in our security models) is able to garner about some hidden value. For example, knowing four bits of a 32-bit value implies that 2^4 of the 2^{32} possible values can be "fixed," leaving only 2^{28} possibilities for the hidden value, e.g. in the remaining 28 bits. Thus we say the adversary has gained four bits of information, or some protocol involved has "leaked" those four bits. However, it is not limited to knowing or not knowing a single bit; *partial* information can also be leaked, implying that the set of total possibility for the hidden value has been reduced even if no individual bits are known to be exclusively zero or one. Leakage is of particular interest in cryptographic protocols where hidden values are intended to be protected from prying eyes. If a protocol leaks too much, adversaries may irreparably gain partial or complete knowledge of these private inputs. In this work we explore leakage which is "inherent" or "intrinsic" to given functionalities, meaning, the leakage on the input which unavoidably occurs as long as the adversary knows the functionality and the corresponding output. Regardless of the robust cryptographic security of a secure protocol, this leakage may compromise private inputs to an undesirable extent.

Corruption thresholds. In secure multi-party computation (MPC) there is a notion of thresholds of corrupted parties (or participants). That is, if a protocol is to be executed amongst n parties, some $t < n$ may be considered to be “corrupted” if they are controlled by the notional adversary to the protocol. Obviously if all n parties are corrupted then no security can be guaranteed. Various cases are studied in the MPC literature, to varying degrees of flexibility on the extents of what can be achieved under given conditions. We will review a number of thresholds which are meaningful. First, $t = n - 1$, where only a single party behaves honestly; in this case some basic protocols may remain secure such as shared randomness generation (“coin-flipping protocols”) which often `XOR` random inputs from participants. As long as one honestly-generated random value is included in the exclusive-or operation, the output will be random. Next, there is $t < \frac{2n}{3}$, where at least $\frac{1}{3}$ of participants remain honest; some functionalities can be made resilient in limited ways against such conditions. Naturally, thresholds of t near $\frac{n}{2}$ are next; whether guaranteeing at least some dishonest or honest majority of minimal size. Particularly in the honest majority case, the range of functionalities and the levels of resilience to attack increase significantly. Finally, $t < \frac{n}{3}$, an honest super-majority of at least two thirds, greatly increases the range and kinds of resilience. In this model, Byzantine fault-tolerant [LSP19; BFM90] protocols can be employed to ensure eventual consistency amongst the honest parties. In all models we also consider whether the adversarial participants are “honest-but-curious,” that is, if they will *behave* honestly but inspect any protocol messages to attempt to ascertain secrets, or “fully malicious,” participating in the protocol but also freely interacting outside the “rules” of the protocol, e.g. by sending invalid messages, attempting to spoof their identities, or selectively rejecting/aborting.

Domain-specific languages. A domain-specific language (DSL) is a programming language designed to encode meaningful operations within a given context, usually as a convenience for practitioners. DSLs are often subsets of or inspired by “fully qualified” programming languages, and generally rely on those languages to provide interpreters or translators to give the DSL purpose. DSLs are a common way for practitioners to encode domain knowledge in a “documentation-as-code” sense, enabling non-expert users to engage with and apply the encoded expertise to achieve some meaningful goals within the domain. In this work we explore DSLs as a method to encode domain expertise about Boolean satisfiability for bitvectors in order to ease the use of the tools we provide; our goal is to enable non-experts to meaningfully encode their target programs/functionalities using our DSLs rather than “raw” Boolean circuits, so that our software tools may intake them and provide useful results.

Chapter 2

Delphinium

This chapter is based on joint work with Gabrielle Beck originally published in Usenix Security 2020 [BZG20a].

There exists extensive practical guidance such as standards [NIS24] and a wealth of research literature on the topics of cryptosystems that are safe, secure, or recommended for use. However, it is often much less clear whether a given system is safe to use *in a given context or setting*, and indeed it is often in the interactions *between* systems where vulnerabilities may arise, rather than in protocol designs themselves.

The past decades have seen enormous improvement in our understanding of cryptographic protocol design. Consider, for example, web encryption techniques: from the earliest versions of SSL [WS+96] to the most recent iterations of TLS [Dow+15], industry and academic communities have worked together to dramatically improve the practice of secure protocol design. Despite these advances, vulnerable protocols remain widely deployed. In many cases this is a result of continued support for legacy protocols and ciphersuites, such as TLS CBC-mode ciphers [Smi12; AP13], export-grade encryption [Beu+15; Adr+15; Avi+16], and legacy email encryption [Pod+18]. However, support for legacy protocols does not account for the presence of vulnerabilities in more recent cryptosystems [W3C17; JS11; Mau+15; Gar+16a; VP17].

In this chapter we consider a specific class of vulnerability: the continued use of unauthenticated symmetric encryption in many cryptographic systems. While the research community has long noted the threat of adaptive-chosen ciphertext attacks on malleable encryption schemes [Bel96; NY90; BN00], these concerns gained practical salience with the discovery of *padding oracle* attacks on a number of standard encryption protocols [Vau02; Hun10; Bar+12; AP13; MDK14; APW09; DP10; Can+03a; Mit05]. Despite repeated warnings to industry, variants of these attacks continue to plague modern systems, including TLS 1.2's CBC-mode ciphersuite [AP13; AP15; Mer+19] and hardware key management tokens [Bar+12; AF18]. A generalized variant, the *format oracle attack* can be constructed when a decryption oracle leaks the result of applying some (arbitrarily complex) format-checking predicate F to a decrypted plaintext. Format oracles appear even in recent standards such as XML encryption [JS11; Kup+15], Apple's iMessage [Gar+16a] and modern OpenPGP implementations [Mau+15; Pod+18]. These attacks likely represent the "tip of the iceberg": many vulnerable systems may remain undetected, due to the difficulty of exploiting non-standard format oracles. We seek to empower end users and non-expert practitioners to uncover vulnerabilities in this class to determine the safety of cryptosystems in the context of their use.

From a constructive viewpoint, format oracle vulnerabilities seem easy to mitigate: simply mandate that protocols use authenticated encryption. Unfortunately, even this advice may be insufficient: common authenticated encryption schemes can become insecure due to implementation flaws such as nonce re-use [Jou06; MW16; Böc+16]. Setting aside implementation failures, the continued deployment of unauthenticated encryption raises an obvious question: *why do these vulnerabilities continue to appear in modern protocols?* The answer highlights a disconnect between the theory and the practice of applied cryptography. In many cases, a vulnerable protocol is not obviously an *exploitable* protocol. This is particularly true for non-standard format

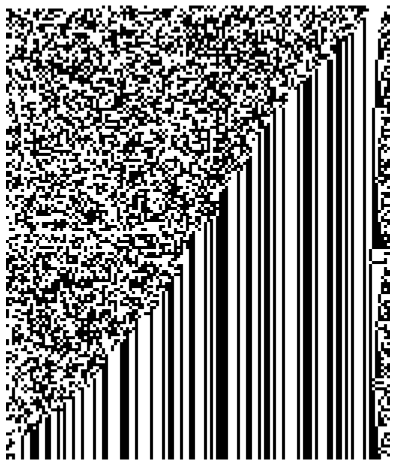


Figure 2.1: Output of a format oracle attack that our algorithms developed against a bitwise padding check oracle $\mathsf{F}_{\mathsf{bitpad}}$ (see §2.10.2 for a full description). The original ciphertext is a valid 128-bit (random) padded message encrypted using a stream cipher. Each row of the bitmap represents a *malleation string* that was exclusive-ORed with the ciphertext prior to making a decryption query.

oracles which require entirely new exploit strategies. As a concrete example, the authors of [Gar+16a] report that Apple did not repair a complex gzip compression format oracle in the iMessage protocol when the lack of authentication was pointed out; but did mitigate the flaw when a concrete exploit was demonstrated. Similar flaws in OpenPGP clients [Gar+16a; Pod+18] and PDF encryption [Mül+19] were addressed only when researchers developed proof-of-concept exploits. The unfortunate aspect of this strategy is that cryptographers' time is limited, which leads protocol designers to discount the exploitability of real cryptographic flaws.

Removing the human (expert) element. In this work we investigate the feasibility of *automating the design and development* of adaptive chosen ciphertext attacks on symmetric encryption schemes. We stress that our goal is not simply to automate the execution of known attacks, as in previous works [Kup+15]. Instead, we seek to develop a methodology and a set of tools to (1) evaluate if a system is vulnerable to practical exploitation, and (2) programmatically derive a novel exploit strategy, given only a description of the target. This removes the expensive human element from attack development, obviating the need for specialized expertise and democratizing the process of determining cryptosystem safety.

To emphasize the ambitious nature of our problem, we summarize our motivating research question as follows:

> *Given a machine-readable description of a format checking function* F *along with a description of the encryption scheme's malleation properties, can we programatically derive a chosen-ciphertext attack that allows us to efficiently decrypt arbitrary ciphertexts?*

Our primary requirement is that the software responsible for developing this attack should require no further assistance from human beings. Moreover, the developed attack must be efficient: ideally it should not require substantially more work (as measured by number of oracle queries and wall-clock execution time) than the equivalent attack developed through manual human optimization.

To our knowledge, this work represents the first attempt to automate the discovery of *novel* adaptive chosen ciphertext attacks against symmetric format oracles. While our techniques are designed to be general, in practice they are unlikely to succeed against every possible format checking function. Instead, in this work we initiate a broader investigation by exploring the limits of our approach against various real-world and contrived format checking functions. Beyond presenting our techniques, our practical contribution includes a tool set that we name `Delphinium`, which produces highly-efficient attacks across several such functions.

Relationship to previous automated attack work. Previous work [BRB18; Pha+17; CSP16] has looked at automatic discovery and exploitation of *side channel* attacks. In this setting, a program combines a fixed secret input with many “low” inputs that are (sometimes adaptively) chosen by an attacker, and produces a signal, *e.g.,* modeling a timing result. This setting can be viewed as a special case of our general model (and vice versa). Like our techniques, several of these works employ SAT solvers and model counting techniques. However, beyond these similarities, there are fundamental differences that manifest in our results: (1) in this work we explore a new approach based on approximate model counting, and (2) as a result of this approach, our results operate over much larger secret domains than the cited works. To illustrate the differences, our experimental results succeed on secret (message) domains of several hundred bits in length, with malleation strings (“low inputs”) drawn from similarly-sized domains. By contrast, the cited works operate over smaller secret domains that rarely even reach a size of 2^{24}. Moreover, our format functions are relatively complex. It is an open question to determine whether the experimental results in the cited works can be scaled using our techniques.

Our contributions.. First, we propose new, and *fully automated* algorithms for developing format oracle attacks on symmetric encryption (and hybrid encryption) schemes. Our algorithms are designed to work with arbitrary format checking functions, using a machine-readable description of the function and the scheme’s malleation features to develop the attack strategy. Second, we design and implement novel attack-development techniques that use approximate model counting techniques to achieve significantly greater efficiency than previous works. These techniques may be of independent interest. Third, we show how to implement our technique practically with existing tools such as SAT and SMT solvers; and propose a number of efficiency optimizations designed to improve performance for specific encryption schemes and attack conditions. Next, we develop a working implementation of our techniques using “off-the-shelf” SAT and SMT packages, and provide the resulting software package (which we call `Delphinium`) as an open source tool for use and further development by the research community. Finally, We validate our tool experimentally, deriving several attacks using different format-checking functions. These experiments represent, to our knowledge, the first evidence of a completely functioning end-to-end machine-developed format oracle attack.

2.1 Intuition

Implementing a basic format oracle attack. In a typical format oracle attack, the attacker has obtained some target ciphertext $C^* = \mathsf{Encrypt}_K(M^*)$ where K and M^* are unknown. She has access to a decryption oracle that, on input any chosen ciphertext C, returns $\mathsf{F}(\mathsf{Decrypt}_K(C)) \in \{0,1\}$ for some known predicate F. The attacker may have various goals, including plaintext recovery and forgery of new ciphertexts. Here we will focus on the former goal.

The challenge in this setting is to programmatically *derive* an attack strategy given only a description of F and encryption scheme. Expressed at a conceptual level, conducting a format oracle attack requires the attacker to formulate a series of *experiments*, each based on the current state of the attacker's knowledge. In our approach, each experiment will comprise a specific *malleation* that can be applied to the target C^*. We must generate these experiments using only a machine-readable description of the format checking function and the malleation features of the scheme.

We define $\mathsf{Maul}_{\mathsf{plain}}$ as a function that, on input a plaintext M and some opaque "malleation instruction string" S, outputs (zero or more) possible "mauled" plaintexts M'. The critical aspect of this function is that the malleation function corresponds to some concrete ciphertext malleation function $\mathsf{Maul}_{\mathsf{ciph}}$ that can apply the same malleation string to a ciphertext, without knowledge of the encryption key.

At the highest level, this attack can be viewed as a repeated process of (1) deriving initial constraints over M, (2) generating an experiment that can be submitted to the oracle, and (3) using the result of this experiment to update the constraint set on M. The attack concludes when the constraints uniquely define M, or when no further constraints can be identified.

Given these inputs, our algorithms must now programmatically (and adaptively) generate a sequence of *experiments*, each of which comprises some malleation instruction string S that can be used to *maul* the target ciphertext. The resulting mauled ciphertext is submitted to the decryption oracle, and the result $b \in \{0,1\}$ is fed back into our algorithm, which completes when M is known or no further experiments can be generated. A key requirement is that our techniques are *general*: *i.e.,* they are not tuned to work with any specific format-checking function or malleation function.

In the remainder of this section we will express our constraint formulae in an "ideal" language that includes clauses such as set membership operations and universal quantifiers. We note that this notation is used purely to simplify the exposition. In practice, theory solvers are significantly more constrained in the logical operations they can reason over. A key technical contribution of this work is showing how to realize our constructions in the face of these limitations.

Describing malleability. Our attacks exploit the malleability characteristics of symmetric encryption schemes. Because the encryption schemes themselves can be complex, we do not want our algorithms to reason over the encryption mechanism itself. Instead, for a given encryption scheme Π, we require the user to develop two efficiently-computable functions that define the malleability properties of the scheme. The function $\mathsf{Maul}^{\Pi}_{\mathsf{ciph}}(C,S) \rightarrow C'$ takes as input a valid ciphertext and some opaque *malleation instruction string* S (henceforth "malleation string"), and produces a new, mauled ciphertext C'. The function $\mathsf{Maul}^{\Pi}_{\mathsf{plain}}(M,S) \rightarrow M'$ computes the equivalent malleation over some plaintext, producing a plaintext or, in some cases, a set of possible plaintexts. This captures the fact that, in some encryption schemes (*e.g.,* CBC-mode encryption), malleation produces *key-dependent* effects on the decrypted message. We discuss and formalize this in §2.2. The essential property we require from these functions is that the plaintext malleation function should "predict" the effects of encrypting a plaintext M, mauling the resulting ciphertext, then subsequently decrypting the result. For some typical encryption schemes, these functions can be simple: for example, a simple stream cipher can be realized by defining both functions to be bitwise exclusive-OR. However, malleation functions may also implement features such as truncation or more sophisticated editing, which could imply a complex and structured malleation string.

Building block: theory solvers. Our techniques make use of efficient theory solvers, such as SAT and Satisfiability Modulo Theories (SMT) [Gan18; MB08]. SAT solvers apply a variety of tactics to identify or rule out a satisfying assignment to a boolean constraint formula, while SMT adds a broader range of theories and tactics such as integer arithmetic and string logic. While in principle our techniques can be extended to work with either system, in practice we will focus our techniques to use quantifier-free operations over bitvectors (a theory that easily reduces to SAT). In later sections, we will show how to realize these techniques efficiently using concrete SAT and SMT packages.

While enormous progress has been made in both the theory and practice of SMT and SAT solving, theory solvers do not, by themselves, provide a solution to our problem. In particular, these tools have no concept of formulating oracle queries to a cryptographic functionality. To realize our techniques, we instead use these solvers as a subroutine in a larger attack algorithm. A fortunate aspect of this approach is that we are thus able to use existing solvers as a "black box," rather than developing new solver theories. This means that our techniques can be instantiated with a variety of off-the-shelf theory solvers [MB08; Gan18; SNC09; Nie+18].

Anatomy of our attack algorithm. The essential idea in our approach is to model each phase of a chosen ciphertext attack as a constraint satisfaction problem. At the highest level, we begin by devising an initial constraint formula that defines the known constraints on (and hence, implicitly, a set of candidates for) the unknown plaintext M^*. At each phase of the attack, we will use our current knowledge of these constraints to derive an *experiment* that, when executed against the real decryption oracle, allows us to "rule out" some non-zero number of plaintext candidates. Given the result of a concrete experiment, we can then update our constraint formula using the new information, and continue the attack procedure until no further candidates can be eliminated.

In the section that follows, we use $\mathcal{M}_0, \mathcal{M}_1$ to represent the partition of messages induced by a malleation string. M_0 and M_1 represent concrete plaintext message assignments chosen by the solver, members of the respective partitions.

The process of deriving the malleation string represents the core of our technical work. It requires our algorithms to reason deeply over both the plaintext malleation function and the format checking function in combination. To realize this, we rely heavily on theory solvers, together with some novel optimization techniques.

Attack intuition. We now explain the full attack in greater detail. To provide a clear exposition, we will begin this discussion by discussing a simplified and *inefficient* precursor algorithm that we will later optimize to produce our main result. Our discussion below will make a significant simplifying assumption that we will later remove: namely, that $\mathsf{Maul}_{\mathsf{plain}}$ will output exactly one plaintext for any given input. This assumption is compatible with common encryption schemes such as stream ciphers, but will not be valid for other schemes where malleation can produce key-dependent effects following decryption.

We now describe the basic steps of our first attack algorithm.

Step 0: Initialization. At the beginning of the attack, our attack algorithm receives as input a target ciphertext C^*, as well as a machine-readable description of the functions F and $\mathsf{Maul}_{\mathsf{plain}}$. We require that these descriptions be provided in the form of a constraint formula that a theory solver can reason over. To initialize the attack procedure, the user may also provide an initial constraint predicate $G_0 : \{0,1\}^n \to \{0,1\}$ that expresses all known constraints over the value of M^*. Here, n represents an upper bound on the length of the plaintext M^*. If we have no *a priori* knowledge about the distribution of M^*, we can set this initial formula G_0 to be trivial.

Beginning with $i = 1$, the attack now proceeds to iterate over the following two steps:

Step 1: Identify an experiment. Let G_{i-1} be the current set of known constraints on M^*. In this first step, we employ the solver to identify a malleation instruction string S as well as a pair of distinct plaintexts M_0, M_1 that each satisfy the constraints of G_{i-1}. Our goal is to identify an assignment for (S, M_0, M_1) that induces the following specific properties on $\mathcal{M}_0, \mathcal{M}_1$: namely, that each message in the pair, when mauled using S and then evaluated using the format checking function, results in a *distinct* output from F. Expressed more concretely, we require the solver to identify an assignment that satisfies the following constraint formula:

$$G_{i-1}(M_0) = G_{i-1}(M_1) = 1 \ \wedge \tag{2.1}$$

$$\forall b \in \{0,1\} : \mathsf{F}(\mathsf{Maul}_{\mathsf{plain}}(M_b, S)) = b$$

If the solver is unable to derive a satisfying assignment to this formula, we conclude the attack and proceed to Step (3). Otherwise we extract a concrete satisfying assignment for S, assign this value to $\mathbf{S}$, and proceed to the next step.

Step 2: Query the oracle; update the constraints. Given a concrete malleation string $\mathbf{S}$, we now apply the ciphertext malleation function to compute an experiment ciphertext $C \leftarrow \mathsf{Maul}_{\mathsf{ciph}}(C^*, \mathbf{S})$, and submit C to the decryption oracle. When the oracle produces a concrete result $\mathbf{r} \in \{0,1\}$, we compute an updated constraint formula G_i such that for each input M, it holds that:

$$G_i(M) \leftarrow (G_{i-1}(M) \ \wedge \ \mathsf{F}(\mathsf{Maul}_{\mathsf{plain}}(M, \mathbf{S})) = \mathbf{r})$$

If possible, we can now ask the solver to *simplify* the formula G_i by eliminating redundant constraints in the underlying representation. We now set $i \leftarrow i+1$ and return to Step (1).

Step 3: Attack completion. The attack concludes when the solver is unable to identify a satisfying assignment in Step (1). In the ideal case, this occurs because the constraint system G_{i-1} admits only one possible candidate plaintext, M^*: when this happens, we can employ the solver to directly recover M^* and complete the attack. However, the solver may also fail to find an assignment because no further productive experiment can be generated, or simply because finding a solution proves computationally intractable. When the solver conclusively rules out a solution at iteration $i = 1$ (*i.e.,* prior to issuing any decryption queries) this can be taken as an indication that a viable attack is not practical using our techniques. Indeed, this feature of our work can be used to rule out the exploitability of certain systems, even without access to a decryption oracle. In other cases, the format oracle may admit only partial recovery of M^*. If this occurs, we conclude the attack by applying the solver to the final constraint formula G_{i-1} to extract a human-readable description of the remaining candidate space (*e.g.,* the bits of M^* we are able to uniquely recover).

Remark on efficiency. A key feature of the attack described above is that it is *guaranteed* to make progress at each round in which the solver is able to find a satisfying assignment to Equation (1). This is fundamental to the constraint system we construct: our approach forces the solver to ensure that each malleation string S implicitly partitions the candidate message set into a pair (M_0, M_1), such that malleation of messages in either subset by S will produce distinct outputs from the format checking function F. As a consequence of this, for any possible result from the real-world decryption oracle, the updated constraint formula G_i *must* eliminate at least one plaintext candidate that satisfied the previous constraints G_{i-1}.

While this property ensures progress, it does not imply that the resulting attack will be *efficient*. In some cases, the addition of a new constraint will fortuitously rule out a large number of candidate plaintexts. In other cases, it might only eliminate a single candidate. As a result, there exist worst-case attack scenarios where the algorithm requires *as many queries as there are candidates for M^**, making the approach completely unworkable for practical message sizes. Addressing this efficiency problem requires us to extend our approach.

Improving query profitability. We can define the *profitability* $\psi(G_{i-1}, G_i)$ of an experimental query by the number of plaintext candidates that are "ruled out" once an experiment has been executed and the constraint formula updated. In other words,

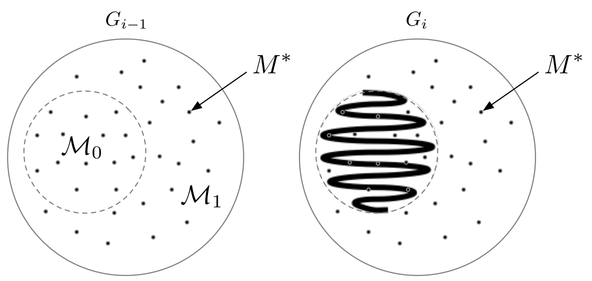


Figure 2.2: Left: illustration of a plaintext candidate space defined by G_{i-1}, highlighting the two subsets $\mathcal{M}_0, \mathcal{M}_1$ induced by a specific malleation string $\mathbf{S}$. Right: the candidate space defined by G_i, in which many candidates have been eliminated following an oracle response $b = 1$.

this value is defined as the number of plaintext candidates that satisfy G_{i-1} but do *not* satisfy G_i. The main limitation of our first attack strategy is that it does not seek to optimize each experiment to maximize query profitability.

To address this concern, let us consider a more general description of our attack strategy, which we illustrate in Figure 2.2. At the i^{th} iteration, we wish to identify a malleation string $\mathbf{S}$ that defines two disjoint subsets $\mathcal{M}_0, \mathcal{M}_1$ of the current candidate plaintext space, such that for any concrete oracle result $\mathbf{r} \in \{0,1\}$ and $\forall M \in \mathcal{M}_{\mathbf{r}}$ it holds that $\mathsf{F}(\mathsf{Maul}_{\mathsf{plain}}(M, \mathbf{S})) = \mathbf{r}$. In this description, any concrete decryption oracle result must "rule out" (at a minimum) every plaintext contained in the subset $\mathcal{M}_{1-\mathbf{r}}$. This sets $\psi(G_{i-1}, G_i)$ equal to the cardinality of $\mathcal{M}_{1-\mathbf{r}}$.

To increase the profitability of a given query, it is therefore necessary to maximize the size of $\mathcal{M}_{1-\mathbf{r}}$. Of course, since we do not know the value $\mathbf{r}$ prior to issuing a decryption oracle query, the obvious strategy is to find $\mathbf{S}$ such that *both* $\mathcal{M}_0, \mathcal{M}_1$ are as large as possible. Put slightly differently, we wish to find an experiment $\mathbf{S}$ that maximizes the cardinality of the smaller subset in the pair. The result of this optimization is a greedy algorithm that will seek to eliminate the largest number of candidates with each query.

Technical challenge: model count optimization. While our new formulation is conceptually simple, actually realizing it involves overcoming serious limitations in current theory solvers. This is due to the fact that, while several production solvers provide optimization capabilities [MB08], these heuristics optimize for the *value* of specific variables. Our requirement is subtly different: we wish to solve for a candidate S that maximizes the *number of satisfying solutions* for the variables M_0, M_1 in Equation (1). Some experimental SMT implementations provide logic for reasoning about the cardinality of small sets, these strategies scale poorly to the large sets we need to reason about in practical format oracle attacks in domains of exponential size.

Unfortunately, this problem is both theoretically and practically challenging. Indeed, merely *counting* the number of satisfying assignments to a constraint formula is known to be asymptotically harder than SAT [Val79b; Tod91], and practical counting algorithms solutions [BL99; BJP00] tend to perform poorly when the combinatorial space is large and the satisfying assignments are sparsely distributed throughout the space, a condition that is likely in our setting. The specific optimization problem our techniques require proves to be even harder. Indeed, only recently was such a problem formalized, under the name Max#SAT [FRS17].

Approximating Max#SAT. While an exact solution to Max#SAT is known to be $\mathsf{NP}^{\mathsf{PP}}$-complete [FRS17; Tod91], several works have explored *approximate* solutions to this and related counting problems [GSS06; CMV16a; SM19; FRS17]. One powerful class of approximate counting techniques, inspired by the theoretical work of Valiant and Vazirani [VV86] and Stockmeyer [Sto83], uses a SAT oracle as follows: given a constraint formula F over some bitvector T, add to F a series of s random parity constraints, each computed over the bits of T. For $j = 1$ to s, the j^{th} parity constraint can be viewed as requiring that $H_j(T) = 1$ where $H_j : \{0,1\}^{|T|} \rightarrow \{0,1\}$ is a universal hash function. Intuitively, each additional constraint reduces the number of satisfying assignments approximately by half, independently of the underlying distribution of valid solutions. The implication is as follows: if a satisfying assignment to the enhanced formula exists, we should be convinced (probabilistically) that the original formula is likely to possess on the order of 2^s satisfying assignments. Subsequently, researchers in the model counting community showed that with some refinement, these approximate counting strategies can be used to approximate Max#SAT [FRS17], although with an efficiency that is substantially below what we require for an efficient attack.

To apply this technique efficiently, we develop a custom count-optimization procedure, and apply it to the attack strategy given in the previous section. At the start of each iteration, we begin by conjecturing a candidate set size 2^s for some non-negative integer s, and then we query the solver for a solution to (S, M_0, M_1) in which approximately 2^s solutions can be found for *each* of the abstract bitvectors M_0, M_1. This involves modifying the equation of Step (1) by adding s random parity constraints to *each* of the abstract representations of M_0 and M_1. We now repeatedly query the solver on variants of this query, with increasing (resp. decreasing) values of s, until we have identified the maximum value of s that results in a satisfying assignment. Note that $s = 0$ represents the original constraint formula, and so a failure to find a satisfying assignment at this size triggers the conclusion of the attack. For a sufficiently high value of s, this approach effectively eliminates many "unprofitable" malleation string candidates and thus significantly improves the efficiency of the attack.

The main weakness of this approach stems from the probabilistic nature of the approximation algorithm. Even when 2^s satisfying assignments exist for M_0, M_1, the solver may deem the extended formula unsatisfiable with relatively high probability. In our approach, this false-negative will cause the algorithm to reduce the size of s, potentially resulting in the selection of a less-profitable experiment S. Following Gomes *et al.* [GSS06], we are able to substantially improve our certainty by conducting *t trials* within each query, accepting if and only if at least $\lceil (\frac{1}{2} + \delta)t \rceil$ trials are satisfied, where δ

is an adjustable tolerance parameter.

Unlike Gomes *et al.* (who only consider model counting), our optimization approach does not allow us to perform these trials over the course of several distinct solver queries, since each trial in our optimization must be bound to the same abstract malleation string S. Thus all trials must be contained within a single extended constraint formula in order to ensure a specific optimal S is found. This added complexity requires us to carefully tune the parameters of our attack, in order to trade query profitability against solver runtime. We discuss these tradeoffs in more detail in §2.10.

Putting it all together. The presentation above is intended to provide the reader with a simplified description of our techniques. However, this discussion does not convey most challenging aspect of our work: namely, the difficulty of implementing our techniques and making them practical, particularly within the limitations of existing theory solvers. Achieving the experimental results we present in this work represents the result of months of software engineering effort and manual algorithm optimization. We discuss these challenges more deeply in §2.8.

Using our techniques we were able to re-discover both well known and entirely novel chosen ciphertext attacks, all at a query efficiency nearly identical to the (optimal in expectation) human-implemented attacks. Our experiments not only validate the techniques we describe in this work, but they also illustrate several possible avenues for further optimization, both in our algorithms and in the underlying SMT/SAT solver packages. Our hope is that these results will inspire further advances in the theory solving community.

2.2 Preliminaries

Notation. Let λ be a security parameter. We will use $A \| B$ to denote string concatenation, and $|A|$ to indicate the length of a string A in bits. Given bitstrings A, B of unequal length, we will denote $A \oplus B$ to be the bitwise exclusive-or of A and B, where the shorter of the two strings is padded on the right side to the length of the longer. When describing constraint formulae, we will use standard typeface to refer to abstract variables, and boldface to indicate a *concrete* variable.

2.2.1 Encryption Schemes and Malleability

Our attacks operate assume that the target system is using a malleable symmetric encryption scheme. We now provide definitions for these terms.

Definition 1 (Symmetric encryption) A symmetric encryption scheme Π is a tuple of algorithms $(\mathsf{KeyGen}, \mathsf{Encrypt}, \mathsf{Decrypt})$ where $\mathsf{KeyGen}(1^{\lambda})$ generates a key, the probabilistic algorithm $\mathsf{Encrypt}_K(M)$ encrypts a plaintext M under key K to produce a ciphertext C, and the deterministic algorithm $\mathsf{Decrypt}_K(C)$ decrypts C to produce a plaintext or the distinguished error symbol $\perp$. We use $\mathcal{M}$ to denote the set of valid plaintexts accepted by a scheme, and $\mathcal{C}$ to denote the set of valid ciphertexts.

Our techniques exploit encryption schemes that are *malleable*, meaning that there exists an efficient ciphertext transformation that induces a predictable effect upon decryption. To derive our attack, we require the user to provide us with a machine-readable description of the malleation features of the scheme.

2.2.1.1 Malleation Functions

The description of malleation functions is given in the form of two functions. The first takes as input a ciphertext along with an opaque data structure that we refer to as a *malleation instruction string*, and outputs a mauled ciphertext. The second function performs the analogous function on a plaintext. We require that the following intuitive relationship hold between these functions: given a plaintext M and an instruction string, the plaintext malleation function should "predict" the effect of mauling (and subsequently decrypting) a ciphertext that encrypts M.

Definition 2 (Malleation functions) The *malleation functions* for a symmetric encryption scheme Π comprise a pair of efficiently-computable functions $(\mathsf{Maul}^{\Pi}_{\mathsf{ciph}}, \mathsf{Maul}^{\Pi}_{\mathsf{plain}})$ with the following properties. Let $\mathcal{M}, \mathcal{C}$ be the plaintext (resp. ciphertext) space of Π. The function $\mathsf{Maul}^{\Pi}_{\mathsf{ciph}} : \mathcal{C} \times \{0,1\}^* \to \mathcal{C} \cup \{\perp\}$ takes as input a ciphertext and a *malleation instruction string*. It outputs a ciphertext or the distinguished error symbol $\perp$. The function $\mathsf{Maul}^{\Pi}_{\mathsf{plain}} : \mathcal{M} \times \{0,1\}^* \to \hat{\mathcal{M}}$, on input a plaintext and a malleation instruction string, outputs a *set* $\hat{\mathcal{M}} \subseteq \mathcal{M} \cup \{\perp\}$ of possible plaintexts (augmented with the decryption error symbol $\perp$). The structure of the malleation string is entirely defined by these functions; since our attack algorithms will reason over the functions themselves, we treat S itself as an opaque value.

We say that $(\mathsf{Maul}^{\Pi}_{\mathsf{ciph}}, \mathsf{Maul}^{\Pi}_{\mathsf{plain}})$ *describes* the malleability features of Π if malleation of a ciphertext always induces the expected effect on a plaintext following encryption, malleation and decryption. More formally, $\forall K \in \mathsf{KeyGen}(1^{\lambda}), \forall C \in \mathcal{C}, \forall S \in \{0,1\}^*$ the following relation must hold whenever $\mathsf{Maul}^{\Pi}_{\mathsf{ciph}}(C,S) \neq \perp$:

$$\mathsf{Decrypt}_K(\mathsf{Maul}^{\Pi}_{\mathsf{ciph}}(C,S)) \in \mathsf{Maul}^{\Pi}_{\mathsf{plain}}(\mathsf{Decrypt}_K(C),S)$$

Simple stream ciphers (no truncation). Stream ciphers employ a pseudorandom generator to produce a keystream that is combined with the plaintext using bitwise exclusive-OR. We will consider an abstract stream cipher Π_{stream}, with the caveat that our formulation omits many practical details such as the generation and delivery of nonces/IVs/state (in this formulation we treat these details as external to the ciphertext). To encrypt a message $M \in \{0,1\}^l$, the $\mathsf{Encrypt}$ algorithm generates a keystream $KS \in \{0,1\}^l$ based on some (unspecified) state or randomness. It formulates the ciphertext as $C = KS \oplus M$. Decryption follows the same steps. Because the decryption process will preserve exclusive-OR operations on the ciphertext, we can define the functions $\mathsf{Maul}_{\mathsf{plain}}^{\Pi_{\mathsf{stream}}}$ and $\mathsf{Maul}_{\mathsf{ciph}}^{\Pi_{\mathsf{stream}}}$ to each be the bitwise exclusive-OR function.

Stream ciphers with truncation. Many stream ciphers permit *plaintext truncation*, in which bits of ciphertext (resp. plaintext) are removed from either the beginning or end of the plaintext string. In some stream ciphers, such as CTR-mode encryption, it is possible to truncate one or more blocks from the input by incrementing the Initialization Vector. Not every stream cipher admits this capability. We can define a malleation function $\mathsf{Maul}_{\mathsf{ciph}}^{\Pi_{\mathsf{tstream}}}$ (resp. $\mathsf{Maul}_{\mathsf{plain}}^{\Pi_{\mathsf{tstream}}}$) that parses S as $S' \| l \| r$, where $l, r \geq 0$ are each decoded as an integer indicating the desired amount to truncate from the high-order (resp. low-order) bits of the plaintext. Truncation is not possible for all stream ciphers, and specific realizations have different limitations. We further discuss these in §2.8. In §2.8.2.1 we discuss the collection of encryption schemes and the details of implementing their associated malleation functions.

Remarks. Note that the function $\mathsf{Maul}_{\mathsf{ciph}}^{\Pi}$ outputs a single ciphertext. By contrast, the plaintext malleation function $\mathsf{Maul}_{\mathsf{plain}}^{\Pi}$ outputs a *set of possible decryption results*. This captures the possibility that certain forms of malleation can induce unpredictable key-dependent effects when ciphertexts are decrypted. A simple example of this phenomenon occurs with CBC-mode encrypted ciphertext, where a single-bit change in the ciphertext can result in pseudorandom output within the corresponding block of plaintext.

Finally, we note that a description of these functions is necessary, but not *sufficient* for our attacks to work. Malleation functions can "overfit" an encryption scheme; for example, one can define a trivial $\mathsf{Maul}^{\Pi}_{\mathsf{plain}}$ such that for every possible input, $\mathsf{Maul}^{\Pi}_{\mathsf{plain}}$ simply outputs the set $\mathcal{M} \cup \{\perp\}$. Similarly, a plaintext malleation function can be formulated even for schemes that are explicitly *non*-malleable, such as authenticated encryption modes. The effectiveness of our attacks depends on both the malleability features of the scheme, and the precision with which the malleation functions describe them.

CTR and OFB mode. CTR and OFB modes use an ℓ-bit block cipher to implement a stream cipher. The modes can therefore be described using using the functions above. However, in the case where an explicit IV is transmitted with the ciphertext, we can define a slightly more powerful malleation function that removes a multiple of ℓ bits from the left side of the ciphertext (resp. plaintext). We can extend the functions for $\mathsf{tstream}$ to support this capability by defining S as $l\|r\|S$, and requiring that $l \ mod \ \ell = 0$. (Plaintext malleation for OFB mode is identical, although ciphertext malleation differs slightly).

CBC mode. In CBC mode encryption, an alteration to the i^{th} ciphertext block will cause meaningful alterations in plaintext block $i+1$ following decryption, while plaintext block i will be replaced with pseudorandom output (excepting for the special case of the Initialization Vector, $i = 0$). CFB is similar, except that changes to block i will emerge in block i, while block $i+1$ will be replaced. To describe CBC, the function $\mathsf{Maul}^{\Pi_{\mathsf{CBC}}}_{\mathsf{plain}}$ can be defined as outputting a *list* of every possible plaintext candidate that results from such a malleation. In §2.8 we describe a more efficient encoding of this output.

2.2.2 Theory Solvers and Model Counting

Solvers take as input a system of constraints over a set of variables, and attempt to derive (or rule out the existence of) a satisfying solution. Modern SAT solvers generally rely on two main families of theorem solver: DPLL [DP60; DLL62a] and Stochastic Local Search [HS04]. Satisfiability Modulo Theories (SMT) solvers expand the language of SAT to include predicates in first-order logic, enabling the use of several theory solvers ranging from string logic to integer logic. Both SAT and SMT (which reduces to SAT) are known to be NP-hard. Thus, these solvers can efficiently solve only a subset of all possible formulae. Our prototype implementation uses a quantifier-free bitvector (QFBV) theory solver. In practice, this is implemented using SMT with a SAT solver as a back-end (in principle, our attacks can be extended to other theories, with some additional work that we describe later in this section). As such, we will described our basic algorithms using a simplified SMT notation over bitvectors of specified lengths. For the purposes of describing our algorithms, we specify a query to the solver by the subroutine $\mathsf{SATSolve}\{(A_1,\ldots,A_N):G\}$ where $A_1,\ldots,A_N$ each represent abstract bitvectors of some defined length, and G is a constraint formula over these variables. Except where explicitly mentioned, we assume that each variable A_i represents a bitvector of some specified length. The response from this call provides one of three possible results: (1) sat, as well as a concrete satisfying solution $(\mathbf{A_1}\ldots,\mathbf{A_N})$, (2) the distinguished response unsat, or (3) the error unknown.

Model counting and Max#SAT. While SAT determines the existence of a single satisfying assignment, a more general variant of the problem, $\#SAT$, determines the *number* of satisfying assignments. A variant known as #SMT generalizes model counting to SMT [CDM17]. Since in this work we focus primarily on problems with discrete solutions (*e.g.,* formulae over boolean and fixed-size integer variables) the

#SMT problem can simply be viewed as a slightly modified version of $\#SAT$.

In the literature this problem is known as *model counting* [Val79b; GSS06; BL99; BJP00; San+04; WS05; BDP03; Cha+15].

In this work we make use of a specific optimization variant of the model count problem, which was formulated as Max#SAT by Fremont *et al.* [FRS17]. In a streamlined form, the problem can defined as follows: given a boolean formula $\phi(X,Y)$ over abstract bitvectors X and Y, find a concrete assignment to X that *maximizes* the number of possible satisfying assignments to Y.The formulation of Fremont *et al.* includes an additional set of boolean variables Z that must also be satisfied, but is not part of the optimization problem. We omit this term because it is not used by our algorithms. Note as well that, unlike Fremont *et al.*, our algorithms are not concerned with the actual *count* of solutions for Y. We will make use of this abstraction in our attacks, with realizations discussed in §2.3.2. Specifically, we define our main attack algorithm in terms of a generic Max#SAT oracle that has the following interface:

$$Max\#SAT(\phi,X,Y) \rightarrow \mathbf{X}$$

Fremont *et al.* provide an algorithm called `MaxCount` that approximately solves the Max#SAT problem, using several approximate sampling and counting algorithms as subroutines [CMV16a; Cha+14a; Cha+15]. Our main attack algorithm can be implemented using `MaxCount` directly. However, using this algorithm in practice is extremely computationally expensive. To address this, we develop a heuristic substitute that substantially improves the efficiency of our attacks.

2.2.3 Format Checking Functions

Our attacks assume a decryption oracle that, on input a ciphertext C, computes and returns $\mathsf{F}(\mathsf{Decrypt}_K(C))$. We refer to the function $\mathsf{F} : \mathcal{M} \cup \{\perp\} \rightarrow \{0,1\}$ as a *format checking function*. Our techniques place two minimum requirements on this function: (1) the function F must be efficiently-computable, and (2) the user must supply a machine-readable implementation of F, expressed as a constraint formula that a theory solver can reason over.

Function descriptions. Our attacks employ SAT and SMT solvers. Because our solver will ultimately need to reason over the implementation of F, we require that its description be provided in a compatible form. For SAT solvers this typically implies conjunctive normal form (CNF) or equivalent notation. SMT solvers allow a richer description using a variety of first-order logic predicates, which may involve variable types such as integers, real numbers and strings. Several industrial SMT solvers additionally provide an interface that allows functions to be described in a high-level languages such as Python. Although our techniques primarily rely on a SAT solver as the back-end for our system, we employ Microsoft's Z3 SMT solver [MB08] as our primary user-facing interface, in part because it provides a more flexible interface with support for various input formats. As a result, the description of F provided to our tools can be given as a Python script, SMT-LIB file [BFT16], or even as a boolean circuit description. Phan *et al.* [Pha+17] demonstrated techniques for deriving such constraint systems automatically from high-level languages such as Java which could be combined with our work.

2.3 Constructions

In this section we present a high-level description of our main contribution: a set of algorithms for programmatically conducting a format oracle attack. First, we provide pseudocode for our main attack algorithm, which uses a generic Max#SAT oracle as its key ingredient. This first algorithm can be realized approximately using techniques such as the `MaxCount` algorithm of Fremont *et al.* [FRS17], although this realization will come at a significant cost to practical performance. To reduce this cost and make our attacks practical, we next describe a concrete replacement algorithm that can be used in place of a Max#SAT solver. The combination of these algorithms forms the basis for our tool `Delphinium`.

2.3.1 Main Algorithm

Algorithm 1 presents our main attack algorithm, which we name `DeriveAttack`. This algorithm is parameterized by three subroutines: (1) a subroutine for solving the Max#SAT problem, (2) an implementation of the ciphertext malleation function $\mathsf{Maul}_{\mathsf{ciph}}$, and (3) a decryption oracle $\mathcal{O}_{\mathsf{dec}}$. The algorithm takes as input a target ciphertext C^*, constraint formulae for the functions $\mathsf{Maul}_{\mathsf{plain}}, \mathsf{F}$, and an (optional) initial constraint system G_0 that defines known constraints on M^*.

This algorithm largely follows the intuition described in §2.1. At each iteration, it derives a concrete malleation string $\mathbf{S}$ using the Max#SAT oracle in order to find an assignment that maximizes the number of solutions to the abstract bitvector $M_0 \| M_1$. It then mauls C^* using this malleation string, and queries the decryption oracle $\mathcal{O}_{\mathsf{dec}}$ on the result. It terminates by outputting a (possibly incomplete) description of M^*. This final output is determined by a helper subroutine `SolveForPlaintext` that uses the solver to find a unique solution for M^* given a constraint formula, or else to produce a human-readable description of the resulting model. Our concrete implementation in §2.8 uses the solver to enumerate each of the known and unknown bits of M^*.

```
Algorithm 1: DeriveAttack
  Input: Machine-readable description of $\mathsf{F}, \mathsf{Maul}_{\mathsf{plain}}$; target ciphertext $C^*$; initial
         constraints $G_0$;
  Output: $M^*$ or a model of the remaining plaintext candidates
  Procedure:
  $i \leftarrow 1$;
  do
        Define $\phi(S, M_0 \| M_1)$ as $[G_{i-1}(M_0) = 1 \wedge$
        $G_{i-1}(M_1) = 1 \wedge \mathsf{F}(\mathsf{Maul}_{\mathsf{plain}}(M_0, S)) = 0 \wedge \mathsf{F}(\mathsf{Maul}_{\mathsf{plain}}(M_1, S)) = 1]$;
      $\mathbf{S} \leftarrow$ Max#SAT ($\phi$, $S$, $M_0 \| M_1$);
      if $S \neq \bot$ then
          $\mathbf{r} \leftarrow \mathcal{O}_{\mathsf{dec}}(\mathsf{Maul}_{\mathsf{ciph}}(C^*, \mathbf{S}))$;
          Define $G_i(M)$ as $[G_{i-1}(M) \wedge (\mathsf{F}(\mathsf{Maul}_{\mathsf{plain}}(M, \mathbf{S})) = \mathbf{r})]$;
          $i \leftarrow i + 1$;
  while $S \neq \bot$;
  return SolveForPlaintext($G_i$);
```

Theorem 2.3.1 *Given an exact Max#SAT oracle, Algorithm 1 maximizes in expectation the number of candidate plaintext messages ruled out at each iteration.*

Remarks. Note that a greedy adaptive attack may not be globally optimal. It is hypothetically possible to modify the algorithm, allowing it to reason over multiple oracle queries simultaneously (in fact, Phan *et al.* discuss such a generalization in their side channel work [Pha+17]). We find that this is computationally infeasible in practice. Finally, note also that our proof assumes an exact Max#SAToracle. In practice, this will likely be realized with a probably approximately correct instantiation, causing the resulting attack to be a probably approximately greedy attack.

2.3.2 Realizing the Max#SAT Oracle

Realizing Algorithm 1 in practice requires that we provide a concrete subroutine that can solve specific instances of Max#SAT. We now address techniques for approximately solving this problem.

Realization from Fremont *et al.* Fremont *et al.* [FRS17] propose an approximate algorithm called `MaxCount` that can be used to instantiate our attack algorithms. The `MaxCount` algorithm is based on repeated application of approximate counting and sampling algorithms [CMV16a; Cha+14a; Cha+15], which can in turn be realized using a general SAT solver. While `MaxCount` is approximate, it can be tuned to provide a high degree of accuracy that is likely to be effective for our attacks. Unfortunately, the Fremont *et al.* solution has two significant downsides. First, to achieve the discussed bounds requires parameter selections which induce practically infeasible queries to the underlying SAT solver. Fremont *et al.* address this by implementing their algorithm with substantially reduced parameters, for which they demonstrate good empirical performance. However, even the reduced Fremont *et al.* approach still requires numerous calls to a solver. Even conducting a single approximate count of solutions to the constraint systems in our experiments could take hours to days, and such counts might occur several times in a single execution of `MaxCount`.

A more efficient realization. To improve the efficiency of our implementations, we instead realize a more efficient optimization algorithm we name `FastSample`. This algorithm can be used in place of the Max#SAT subroutine calls in Algorithm 1. Our algorithm can be viewed as being a subset of the full `MaxCount` algorithm of Fremont *et al.*

The `FastSample` algorithm operates over a constraint system $\phi(S, M_0 \| M_1)$, and returns a concrete value $\mathbf{S}$ that (heuristically) maximizes the number of solutions for the bitvectors M_0, M_1. It does this by first conjecturing some value s, and sampling a series of $2s$ low-density parity hash functions of the form $H : \{0,1\}^n \to \{0,1\}$ (where n is the maximum length of M_0 or M_1). It then modifies the constraint system by adding s such hash function constraints to each of M_0, M_1, and asking the solver to find a solution to the modified constraint system. If a solution is found (resp. not found) for a specific s, `FastSample` adjusts the size of s upwards (resp. downwards) until it has found the maximal value of s that produces a satisfying assignment, or else is unable to find an assignment even at $s = 0$.

The goal of this approach is to identify a malleation string **S** as well as the largest integer s such that at least 2^s solutions can be found for each of M_0, M_1. To improve the accuracy of this approach, we employ a technique originally pioneered by Gomes *et al.* [GSS06] and modify each SAT query to include multiple *trials* of this form, such that only a fraction $\delta + 1/2$ of the trials must succeed in order for **S** to be considered valid. The parameters t, δ are adjustable; we evaluate candidate values in §2.10.

Unlike Fremont *et al.* (at least, when implemented at full parameters) our algorithm does not constitute a sound realization of a Max#SAT solver. However, empirically we find that our attacks using `FastSample` produce query counts that are close to the optimal possible attack. More critically, our approach is capable of identifying a candidate malleation string in seconds on the constraint systems we encountered during our experiments.

2.3.3 Additional Algorithms

Our algorithms employ an abstract subroutine `AdjustSize` that is responsible for updating the conjectured set size s in our optimization loop:

$$(b_{\mathsf{continue}}, s', Z') \leftarrow \mathtt{AdjustSize}(b_{\mathsf{success}}, n, s, Z)$$

Algorithm 2: `FastSample`

```
Input: $\phi$ a constraint system over abstract bitvectors $S, M_0 \| M_1$; $n$ the
       maximum length of (each of) $M_0, M_1$; $m$ the maximum length of $S$; $t$
       number of trials; $\delta$ fraction of trials that must succeed
Output: $\mathbf{S} \in \{0,1\}^m$
Procedure:
$(b_{\text{continue}}, s, Z) \leftarrow$ AdjustSize(False, $n, \perp, \varepsilon$);
// define $t$ symbolic copies of the abstract bitvectors $M_0, M_1$, and a
   new constraint system $\phi^t$
$\{M_{1,0}, \ldots, M_{t,0}\} \leftarrow M_0$;
$\{M_{1,1}, \ldots, M_{t,1}\} \leftarrow M_1$;
Define $\phi^t(S, \{M_{1,0}, \ldots, M_{t,0}\}, \{M_{1,1}, \ldots, M_{t,1}\})$ as $\phi(S, M_{1,0} \| M_{1,1}) \wedge \cdots \wedge \phi(S, M_{t,0} \| M_{t,1})$;
while $b_{\text{continue}}$ do
    // Construct $2t$ $s$-bit parity constraints
    for $i \leftarrow 1$ to $t$ do
        $\mathcal{H}_{i,0} \leftarrow$ ParityConstraint $(n, s)$;
        $\mathcal{H}_{i,1} \leftarrow$ ParityConstraint $(n, s)$
    // Query the solver
    $\mathbf{S} \leftarrow$ SATSolve$\{(S, \{M_{1,0}, \ldots, M_{t,0}\}, \{M_{1,1}, \ldots, M_{t,1}\}) :$
        $\exists \mathcal{R}_0 \subseteq [1,t] : |\mathcal{R}_0| \geq \lceil (0.5+\delta)t \rceil, \forall j \in \mathcal{R}_0 : \mathcal{H}_{j,0}(M_{j,0}) = 1 \wedge$
        $\exists \mathcal{R}_1 \subseteq [1,t] : |\mathcal{R}_1| \geq \lceil (0.5+\delta)t \rceil, \forall j \in \mathcal{R}_1 : \mathcal{H}_{j,1}(M_{j,1}) = 1 \wedge$
        $\phi^t(S, \{M_{1,0}, \ldots, M_{t,0}\}, \{M_{1,1}, \ldots, M_{t,1}\})\}$;
    if $S ==$ unsat then
        $b_{\text{success}} =$ False;
    $(b_{\text{continue}}, s, Z) \leftarrow$ AdjustSize($b_{\text{success}}, n, s, Z$);
return $\mathbf{S}$
```

The input bit b_{success} indicates whether or not a solution was found for a conjectured size s, while n provides a known upper-bound. The history string $Z \in \{0,1\}^*$ allows the routine to record state between consecutive calls. `AdjustSize` outputs a bit b_{continue} indicating whether the attack should attempt to find a new solution, as well as an updated set size s'. If `AdjustSize` is called with $s = \perp$, then s' is set to an initial set size to test, $b_{\text{continue}} = \text{True}$, and $Z' = Z$. By maximizing the conjectured set sizes, the attack elicits a malleation string at each iteration which maximizes the number of messages eliminated *in expectation*. As such, this algorithm is greedy, optimizing given the constraints at each iteration to discover locally optimal malleations

The subroutine `ParityConstraint`(n,l) constructs l randomized parity constraints of weight k over a bitvector $b = b_1 b_2 \dots b_n$ where $k \leq n$ denotes the number of bit indices included in a parity constraint (i.e. the parity constraints come from a family of functions $H(b) = \bigoplus_{i=1}^{n} b_i \cdot a_i$ where $a \in \{0,1\}^n$ and the Hamming weight of a is k).

Finally, to output intermediate and finalized results from the solver, we use a widely-known technique in SAT solving to iteratively identify symbolic bits which are entirely constrained to either 0 or 1. For completeness, we describe this technique as `FindKnownBits` in Algorithm 3.

Algorithm 3: `FindKnownBits`

```
Input: Constraint formula $G$, sat oracle SATSolve, bv an $n$ bit length bitvector
Output: $m \in \{0,1,*\}^n$ where $m_i = b$ for $b \in \{0,1\}$ if the bit vector $bv$ at position $i$ is
        uniquely constrained by $G$ [i.e. $\neg\exists$ model $v$ such that $v_i = 1 - b$ and
        $G(v) = 1$] Otherwise $m_i = *$
Procedure:
$m := 0^n$;
for $i \leftarrow 0$ to $n$ do
    $one, \_ = \mathsf{SATSolve}(\{\mathrm{bv}\} : G \wedge [\mathrm{bv}_i \neq 1])$;
    $zero, \_ = \mathsf{SATSolve}(\{\mathrm{bv}\} : G \wedge [\mathrm{bv}_i \neq 0])$;
    if $one$ is unsat then
        $m[i] = 1$;
    else if $zero$ is unsat then
        $m[i] = 0$;
    else
        $m[i] = *$;
return $m$;
```

2.4 Proof that `DeriveAttack` is Greedy

In this section, we demonstrate that our approach implements a locally-optimal (also known as “greedy”) algorithm at each iteration of an attack, given a Max#SAToracle in place of probabilistic subroutines. This idealized model is used to clarify proving; in practice, the probabilistic and approximate nature of these subroutines allows us to pragmatically evade the inherent algorithmic complexity of Max#SAT.

2.4.1 Preliminaries

We prove greedy-optimality by defining optimality with respect to profitability, and showing that at each iteration this function is maximized in expectation. Recall #SAT, an algorithm which given a vector of boolean variables and a constraint formula over those variables returns the number of assignments to the variables satisfy the formula. `DeriveAttack` does not directly use a #SAT oracle, but we use it in this proof as notational shorthand. Max#SAT(X,Y,G) given boolean variable vectors X,Y and constraint formula G provides an assignment for X which satisfies G, maximizing the number of satisfying assignments to Y. That is, no other satisfying assignment for X induces a greater number of satisfying assignments for Y subject to G. Also, recall profitability $\psi(G_{i-1},G_i)$, the number of candidate messages eliminated via additional constraint added between G_{i-1} and G_i. This is equal to $\#SAT((M_0,M_1),G_{i-1})-\#SAT((M_0,M_1),G_i)$. The new constraint depends upon the $\mathcal{O}_{\mathsf{dec}}$ oracle result, and so we will seek to maximize it in expectation quantified over the possible oracle responses $\{0,1\}$. In expectation, profitability is equal to the number of candidate messages eliminated by each possible oracle response multiplied by the probability the oracle provides that result. As the true plaintext message is uniformly sampled, this probability is equal to the number of messages not eliminated by the respective oracle result. Similarly, this approach generalizes to non-uniform distributions using weighted counting.

Let M be the distribution of plaintext messages of some fixed length n. We consider the uniform message distribution, but non-uniform distributions are equivalently supported through weighted model counting (which reduces to unweighted model counting efficiently) [Cha+15]. The message length is known to the attack algorithm.

Let $\mathcal{O}_{\mathsf{dec}}$ be the decryption oracle. `DeriveAttack` minimizes the number of queries to the oracle, making one per greedy iteration. One query must be made per iteration as the oracle provides the only available information about the target plaintext; any other computation in an iteration is necessarily speculative.

Let G_0 be the initial constraint system, a conjunction of the following constraints along with any optional auxiliary constraints:

- $\mathsf{F}(\mathsf{Maul}_{\mathsf{plain}} M_0, S) = 0$
- $\mathsf{F}(\mathsf{Maul}_{\mathsf{plain}} M_1, S) = 1$

G_0 primarily enforces that each assignment $\mathbf{S}$ to S malforms all assignments to M_0 such that they are invalid under the format check F, and simultaneously malforms all assignments to M_1 such that they remain or become valid. We assume that F and $\mathsf{Maul}_{\mathsf{plain}}$ are well-defined, and as such each assignment $\mathbf{S}$ indexes (potentially non-uniquely) some partition of the remaining valid assignments to M_0 and M_1. The additional constraints may include weighting the message distribution, or arbitrary additional requirements for the generated attack.

Let G_i $i > 0$ be defined as G_0 conjoined with i additional constraints. These constraints are of the form

$$\mathsf{F}(\mathsf{Maul}_{\mathsf{plain}}(M_0, \mathbf{S}_i)) = \mathsf{F}(\mathsf{Maul}_{\mathsf{plain}}(M_1, \mathbf{S}_i)) = \mathcal{O}_{\mathsf{dec}}(\mathsf{Maul}_{\mathsf{ciph}}(C, \mathbf{S}_i))$$

with $\mathbf{S}_i$ some assignment to S at each iteration i. This models updating the constraint formula to reflect the result of an $\mathcal{O}_{\mathsf{dec}}$ oracle query. We refer to the constraint added at iteration i as $\mathsf{iter}_{\mathbf{S}_i}$.

2.4.2 Proof

The proof will proceed using strong induction. However, the strong induction case parallels the base case with minimal differences. Thus, the base case will be explored thoroughly, and then in the strong induction case the aforementioned differences will be highlighted and considered. Finally, the case when Max#SATaborts will be considered.

2.4.2.1 Base Case

Isomorphism of concatenation to multiplication: As M_0 and M_1 partition the remaining candidate messages,

$$\#SAT(M_0 \| M_1, G_0) = (\#SAT(M_0, G_0))(\#SAT(M_1, G_0))$$

As each element of the combined count can be bijectively mapped to one choice of assignment to M_0 and one choice of assignment to M_1.

As F is well-defined, no satisfying assignment for M_0 is also a satisfying assignment for M_1. The disjunction of satisfying assignment constitutes all satisfying assignments for the plaintext message. Consider the size of this disjunction to be $x \leq 2^n$. Note that M_0 and M_1 must each have at least one satisfying assignment for the formula to be satisfiable, and at most $x-1$ satisfying assignments.

$Max\#SAT(S, M_0 \| M_1, G_0)$ produces an assignment $\mathbf{S}$ for S which maximizes the number of assignments to $M_0 \| M_1$, thus maximizing the previously described product. Maximizing such a product of two integers in the range $[1, x)$ which sum to x occurs when each integer is as close to $\frac{x}{2}$ as possible. Thus, $\mathbf{S}$ induces as close as possible to $\frac{x}{2}$ satisfying assignments to each of M_0 and M_1.

Next, let $p = \frac{min(\#SAT(M_0,G_0),\#SAT(M_1,G_0))}{x}$. p then represents without loss of generality the fraction of x which the smaller of M_0, M_1 contains. As the message distribution is uniform, p is the likelihood that the plaintext message lies in the smaller set of satisfying assignments, in which case $(1-p)x$ messages will be ruled out. Similarly, the larger set of satisfying assignments contains the plaintext message with probability $1-p$, and in that case px messages will be eliminated. In expectation, the number of eliminated messages:

$$p(1-p)x + (1-p)px = 2xp(1-p)$$

$p \in (0,1)$ as both sets of satisfying assignments are non-empty. This quadratic expression is maximized at $p = 0.5$, consistent with maximizing the product. Let $G_1 = G_1 \wedge \mathsf{iter}_{\mathbf{S}}$.

We seek to demonstrate that without consulting $\mathcal{O}_{\mathsf{dec}}$ (as only one query is allowed per iteration, to minimize oracle queries), $\mathbf{S}$ represents the greedy decision: maximizing the number of eliminated candidates (and thus profitability) in expectation.

Suppose seeking contradiction that some $\mathbf{S}' \neq \mathbf{S}$ exists for which $G_1{}' = G_0 \wedge \mathsf{iter}_{\mathbf{S}'}$ and $\psi(G_0, G_1{}') > \psi(G_0, G_1)$. Define p' similarly, the fraction represented by the smaller of the two satisfying assignment sets. $p' > p$ by necessity, as p' must be closer to 0.5. This implies that the alternate assignment to S induces a more even (close to $\frac{1}{2}$) division of satisfying assignments amongst M_0 and M_1. However, this necessarily implies that $\mathbf{S}'$ induces a larger number of assignments to $M_0 \| M_1$, which violates the correctness of Max#SAT. As such, a contradiction is reached, and so in the base case, profitability is maximized in expectation. Note that nothing in the preceding reasoning specifically relied on the size x or the auxiliary contents of G_0.

2.4.2.2 Strong Induction

Now, assume that for some number of iterations I, `DeriveAttack` generated an assignment for S which eliminated the maximal number of messages in expectation. G_I represents the constraint system at this iteration of the attack.

To complete induction for iteration $I+1$, simply observe that the proof for the base case allowed arbitrary additional constraints within G_0, and did not rely on the overall number of satisfying assignments x. As such, the reasoning is entirely compatible in the inductive case, replacing G_0 with G_I.

2.4.2.3 Max#SAT Aborts

Max#SATmay abort. This implies no satisfying assignment for X or Y can be found, implying either that the formula is inconsistent or that $\mathsf{Maul}_{\mathsf{plain}}$ cannot induce a differentiation in the format check F, meaning no further messages may be eliminated. In either of these cases, `DeriveAttack` should abort, as no further progress can be made.

2.5 The Greedy Algorithm can be Optimal

In the following proof sketch, we justify our greedy algorithm by demonstrating existence of a simple format and malleation function pair for which the greedy attack is optimal. The defined format and malleation pair serves as an existence proof. The intuition behind this format is that each possible query leaks one bit of information by diving the remaining candidate message space in half, every query must be made to uniquely identify the plaintext, and repeat queries do not provide any information whatsoever. As such, both the greedy and optimal attacks are clear, and as is demonstrated, equivalent.

Lemma 2.5.1 *For the format* F *and malleation* $\mathsf{Maul}_{\mathsf{plain}}$ *pair* $(\mathsf{F}_{\mathsf{Parity}}, \mathsf{Truncation})$*, the greedy attack is optimal.*

2.5.1 Preliminaries

Consider the message distribution of uniform k-bit strings. The size of this message space is then 2^k. Consider the format check $\mathsf{F}_{\mathsf{Parity}}$ to be the bitwise parity function, which maps bitstrings of positive length to $\{0,1\}$. A parity value of 1 corresponds to passing the format check, and parity 0 to failing. We will consider the parity of strings of less than 1 bit in length to be undefined.

We will define a plaintext malleation function that performs simple truncation from the right side (the least significant bits) of the plaintext. Such a function can be constructed for stream ciphers, by simply performing the identical truncation on a ciphertext. We only consider right-truncations because left-truncations cause a stream-cipher message to be misaligned with the keystream, leading to unpredictable results from which the attack can extract no information. The malleation string S represents the truncation instruction as a non-negative integer that indicates how many bits will be removed. Truncation values that exceed the message length result in undefined output.

For a given malleation string S (truncation value), let $\mathcal{M}_0$ be the subset of plaintexts where which, $\forall M \in \mathcal{M}_0$, it holds that $\mathsf{F}_{\mathsf{Parity}}(\mathsf{Maul}_{\mathsf{plain}}(M,S)) = 0$, and let $\mathcal{M}_1$ be the equivalent set where the output of the check is 1. Recall profitability $\psi(G_{i-1},G_i)$ from the attack definition, the number of messages "ruled out" at each iteration. In this proof sketch, we operate over a message distribution rather than a single concrete message, and as such *expected* profitability is used for average-case analysis. In this formulation, the most profitable malleation strings in expectation are those which divide the candidate message evenly: where approximately as many messages are in $\mathcal{M}_0$ as in $\mathcal{M}_1$.

Let M^* be the a target plaintext message, with $\mathsf{F}_{\mathsf{Parity}}(M^*) = 1$ and C^* its bitwise stream cipher encryption.

2.5.2 Observations

The 0-length truncation will always return 1 (parity check success), as the original message is defined to be validly formatted and is left unchanged. So, we need only consider the other $k-1$ truncations, those of non-zero length (as you cannot drop k or more bits from a k-bit message).

Assuming the underlying message is uniformly sampled from the k-bit message space, each truncated bit will change the parity with $\frac{1}{2}$ probability. The length-1 truncation then will result in two message distributions of equal size: one where the truncated bit was 1 and the other where it was 0. If the truncated bit was 1, necessarily the parity of the remaining bits is 0, thus the format check over these will fail, and otherwise the parity of the remaining bits is 1 and the format check will pass. The oracle query on this truncation then uniquely determines the truncated bit of M by revealing the parity of the remaining bits.

These message distributions are of size 2^{k-2} each (not $k-1$: the un-truncated unknown bits must have parity 0 or 1 as determined by the contents of the truncated bit), are disjoint, and under union form the entire 2^{k-1} valid message space.

Truncating additional bits in the first query has a similar effect. Consider truncating two bits in the first query: rather than two cases, there are four (the four values of two truncated bits): two cases which leave the un-truncated bits valid (unchanged parity), and two which flip parity. A partition is formed over messages ending in 01 or 10 and messages ending in 00 or 11, and these two message classes are of equal size 2^{k-3}: parity 1 messages of length $k-2$ and parity 0 messages of length $k-2$.

Based on the previous definition of profit, the one- and two- bit truncations are of equal profitability. It's clear that longer truncations follow the same pattern inductively, each creating a partition where both sides consist of twice as many classes of messages where the classes themselves are $\frac{1}{2}$ the size. If all possible truncations are of the same profitability, it must be the case that any choice of truncation is greedy (highest-profitability-first) for the first query in the attack.

2.5.3 The Optimal Attack

There are $k-1$ possible truncations. Repeating a truncation provides information which the attack already knows (the parity of the un-truncated bits) and therefore no optimal attack can repeat a truncation.

Assume some optimal (in terms of query count) attack makes less than $k-1$ queries. Necessarily, some truncation length must be left un-queried. Let that length be u. The attacker must then know the parity of all subsets of bits of length 1 up to $u-1$ from the left. If they know the parity of the leftmost bit, they know the value of the bit. If they know the leftmost bit, and they know the parity of the two leftmost bits, they know the two leftmost bits. Inductively, the attacker knows all bits up to but not including u. The attacker has also made queries on truncations of length $u+1$ up to $k-1$ (if there are any, u might equal $k-1$). Thus, they know the parity of substrings starting from bit u (e.g. u through $u+1$, u through $u+2$, ...). Incrementally inducting up to k, the attacker knows all bits from $u+2$ to k (if any). Importantly, the only information the attacker lacks is the parity of the substring of bits from 1 to u, which prevents them from uniquely constraining bits u and $u+1$. They know the parity of these two bits, and as such only two of the possible four two-bit strings are valid candidates, but there is insufficient information to differentiate them.

The above analysis has edge cases at very small values of k. For example, at $k=1$ or $k=2$, it's unclear where the un-queried index would be. However, fewer than $k-1$

queries in those cases is zero queries, from which no information can possibly be extracted. $k = 1$ is trivial as M must have parity 1, and $k = 2$ clearly has two candidate messages before any queries are made. At $k = 2$, the only valid query is to drop the first bit. The result of this query uniquely constrains the message.

Therefore, after any $k-2$ optimal queries, the attack is left with two possible messages, and thus has not uniquely constrained the message space. This attack then, is incomplete, and thus the assumption that an optimal attack with fewer than $k-1$ queries exists is faulty.

It is clear that querying the un-queried truncation at u would finish the attack, uniquely constraining the message bits in $k-1$ queries by revealing the parity of the bit at u. Thus, any optimal attack must use at least $k-1$ queries, and an optimal attack exists using $k-1$ queries.

2.5.4 The Greedy Attack

As was shown, the optimal attack makes a truncation query at each available length for a total of $k-1$ queries. Order is irrelevant as any $k-2$ optimal queries allow the last query to uniquely constrain the message. In the observations considered prior, initially, all truncations are of equal profitability, and thus the greedy attack could select any of them.

In order to assert that the greedy attack is optimal then, it is sufficient to show that any query made by the greedy attack leads to a state wherein the greedy attack will not make a redundant query. Once $k-1$ non-redundant queries are made, an optimal attack has necessarily been executed.

The profitability of a redundant query is zero at any stage of the attack. This is because, having previously made a given query, the greedy attack excludes from consideration all messages which differ from the oracle at that truncation length. It does so though iterative constraints which assert that the candidate messages behave as the oracle did given a particular malleation. Thus, any redundant query will reject all candidate messages if the oracle returned false on that malleation, or will accept all candidate messages if the oracle returned true. Either way, no messages are differentiated, and so no redundant query can be selected in the greedy attack. Therefore, the greedy attack is optimal.

2.5.5 Discussion

In this simple scenario, the greedy attack is optimal. This can be clearly shown in that all available malleations are either optimal or completely redundant. In real format checks, parity checks sometimes occur, but often more complicated systems of constraints are expressed such as range statements, conditionals, or regular expressions.

2.6 The Greedy Algorithm can be Suboptimal

In the following proof sketch, we provide a counterexample to the universal optimality of the greedy algorithm, as both illuminating analysis of limitations of the attack algorithm and to highlight opportunity for future work. Our greedy attack algorithm operates over a pair consisting of a format checking function and a malleation function, and as such an existence proof outlining the possibility of non-optimality requires definition of each element of such a pair. Here the format check F will be defined as a conditional function given below, and the malleation $\mathsf{Maul}_{\mathsf{plain}}$ as the composition of truncation and exclusive-OR. Thus, The attack algorithm then can flip any subset of bits, and truncate the message down to any non-zero length.

$$\mathsf{F}_{\vee}(M) = \begin{cases} Parity(M_{0..L-(k+1)}) & \text{if } M_{L-(k+1)..L} = cookie \\ Padding(M) & \text{otherwise} \end{cases}$$

This format check is somewhat contrived to "lead astray" the greedy algorithm. Greedy algorithms are non-optimal when a locally non-optimal choice enables higher-efficiency (optimality, or profitability per query) choices later in execution. In order to realize this disparity, we define $\mathsf{F}_{\vee}$ conditionally, where one case applies a parity format check if a cookie value is matched, and the other simply applies a PKCS7-like padding check if the cookie value is not matched. Guessing the cookie value is locally non-optimal, but then the enabled attack against parity is more efficient than the greedy attack against the padding format on average.

Lemma 2.6.1 *For the format and malleation pair* ($\mathsf{F}_{\vee}$, *Truncation* $\circ$ *Exclusive-OR*)*, the greedy attack is not optimal.*

2.6.1 Preliminaries

Consider the message distribution of uniform L-bit strings. The size of this message space is then 2^L. Let $kl = L$ and we require that $2^k > l$. We will consider l chunks of size k for notational simplicity when analyzing this distribution.

Let the encryption scheme be a stream cipher which uses $\oplus$ (exclusive-OR) to mix the message with the keystream, and assume it operates "left to right" (that is, we refer to the beginning of the keystream and the message as the "left side"). Consider the malleation in question to be arbitrary composition of truncation and exclusive-OR, allowing the attack algorithm to flip any subset of bits of the message. This is a commonly available malleation for unauthenticated stream ciphers (or those where authentication can be forged, as is the case in Galois-Counter Mode with nonce reuse). Although truncated messages are potentially accepted by the format, the length of the message is leaked by the known ciphertext and as such the initial candidate message space excludes shortened messages.

Recall profitability $\psi(G_{i-1}, G_i)$ from the attack definition, the number of messages “ruled out” at each iteration. In this proof sketch, we operate over a message distribution rather than a single concrete message, and as such *expected* profitability is used for average-case analysis.

Let M be the real plaintext message, and C its bitwise stream cipher encryption. We allow M to be any L-bit string, not just those which are valid under the predicate F.

When checking a message, F_V considers the leftmost $k+1$ bits, comparing them to some per-instance randomly sampled “cookie” value. Although the attack algorithm has a full description of the format check, extending the model of the oracle to contain private randomness is in no way unaligned with reality. If the top $k+1$ bits match this cookie, the remaining $L-(k+1)$ bits are checked using a bitwise parity check. Otherwise, the entire L-bit message is checked with More intuitive descriptions of each branch of the conditional format check are provided next.

2.6.1.1 Parity Check

This check evaluates the remaining bits after any truncations (at least 1 bit, per the format definition), and returns the parity of those bits. Parity 1 is accepted by the format, parity 0 is rejected. As noted in the greedy-optimal analysis, this format in combination with truncation can leak 1 bit of information per query, allowing efficiency of message discovery linear in the number of bits.

2.6.1.2 Padding Check

This check evaluates L bits in k-bit chunks. As such, any truncation renders a message invalid. The message must be padded with a number of k-bit chunks matching the value of the last chunk. This is very similar to PKCS7, but chunks are not required to be 8 bits. Requiring $2^k > l$ ensures that the padding can stretch over the entire L-bit message. Attacking this format involves malleating chunks of the underlying message to become valid padding, thus creating an exclusive-OR equation with one unknown (which thus can be immediately solved). This requires guessing the correct exclusive-OR malleation string to flip bits in the message to become valid padding. As will become clear, guessing a padding chunk will be more profitable than guessing the cookie value, but enable a less efficient attack than that allowed once the cookie value is discovered in the optimal attack.

2.6.2 Observations

Note that we will primarily consider truncations off the right side of the known ciphertext as this will allow the remaining bits to decrypt as expected. If the keystream is pseudorandom, truncation off the left will misalign the key and ciphertext streams, causing unpredictable bits to arise in the resulting plaintext. As such, the attack algorithm would be unable to differentiate oracle results on such queries from oracle results on random strings, and thus these queries yield trivial information about the target plaintext.

2.6.3 The Optimal Attack

The cookie value is embedded in a conditional as part of the description of the format check provided to the attack algorithm. This analysis still holds in the case that this value is initially unknown to the attack algorithm and it has to guess a $k+1$-bit exclusive-OR string to transform the plaintext into the cookie value in a sequence of queries, but that assumption complicates the model of the format oracle and as such is elided.

Thus, the optimal strategy is to truncate the known ciphertext to $k+2$ bits (any non-zero truncation leaving at least one non-cookie bit is equally effective, but complicates the attack slightly) and guess the plaintext underlying the first $k+1$ bits. As the cookie is known, and the exclusive-OR malleation can be controlled, this requires at most 2^{k+1} queries, or 2^k in expectation. Once a query passes the format check, the cookie is discovered.

However, it may be the case that the parity of the un-truncated, non-cookie bit(s) is 0, which would cause a false oracle query result for any guess of the cookie. By querying each guess twice, once with the un-truncated bit(s) untouched, and once with a single un-truncated bit flipped (exclusive-OR with 0 or 1, respectively), the cookie will be discovered in 2^{k+1} queries in expectation.

Then, the remaining $L-(k+1)$ bits can be discovered one query at a time by truncating to incrementally increasing sizes, at which point each query result describes whether the included bit did or did not change the parity of the string. This completely compromises the plaintext one bit at a time.

Before the cookie is known, the profitability of making a guess is $2^{L-(k+2)}$, as each pair of queries (flipping the un-truncated bits, and not) eliminates the entire class of messages that would have become valid had the guess been correct (which is of size $2^{L-(k+1)}$). After the cookie is known (which also includes uncovering the un-truncated bit) the profitability is half of the remaining messages at each step, maximizing profitability in expectation. The optimal attack in expectation will take 2^{k+1} queries to guess the cookie and the un-truncated bit, and then $L-(k+2)$ for the remaining bits. Thus, expected profitability $\approx 2^{L-(k+2)}$ and expected queries $= 2^{L-(k+1)} + L-(k+2)$.

2.6.4 The Greedy Attack

On the other hand, the greedy attack strategy simply seeks to maximize the profitability of the malleation selections. By attempting to guess a k-bit exclusive-OR string which transforms the last k bits into a valid padding chunk (the k-bit representation of 1, for example), and then proceeding chunk-wise through the l chunks of the messages, the greedy attack will take in expectation $l(2^{k-1})$ queries. That is, each chunk requires 2^{k-1} guesses in expectation, and the attack must guess all l chunks in this way, malleating them via exclusive-OR into increasingly long valid paddings.

The greedy attack will follow this attack path because doing so differentiates a class of messages which match the guess of the last k bits from the remaining which have some other last k bits. As the message is uniform, the number of eliminated messages will likely be the smaller class, of size 2^{L-k}, although with some probability (2^{-k}) it will be the larger class (if the guess was correct). Thus the profitability in expectation is

$$\text{exp. profitability} = (2^{-k})(2^{L} - (2^{L-k})) + (1 - 2^{-k})(2^{L-k})$$

The greedy expected profitability exceeds the optimal expected profitability for all $L > 0$, and thus the greedy attack will attack the padding scheme as described. As the attack progresses, the profitability is exponentially reduced as some number of bits have become known. However, the greedy attack will not switch to attacking the cookie value, because the profitability of a query guessing the cookie is also reduced by these known bits. They decrease at the same exponential rate (each known bit halving the size of the candidate message space) and as such the padding attack remains more profitable.

The greedy attack must guess each k-bit chunk and so:

$$\text{exp. queries} = (2^{k-1})l$$

The expected number of queries for the greedy attack strictly exceeds the expected number of queries for the optimal attack for some choices of L and k. As such, there exists a configuration for which the greedy attack is not optimal.

2.6.5 Discussion

Although this particular format may seem contrived, conditionals in format checks are not rare. Many formats contain headers which define how the remaining message will be processed; examples include indicators used for compression algorithm selection or format version [RP; Pfe03].

2.6.5.1 Edge Cases

There are numerous edge cases due to the conditional nature of the format check. However, as this is an existence proof, many of them can be mitigated through selection of k, L, and l (the selection of any two fixes the third) to reduce the probability and/or impact of the edge case. These cases are summarized below, regardless.

If the two formats overlap, i.e. the cookie bits are also part of a validly padded message, the implication of an oracle result can be unclear. However, by truncating when attacking the cookie bits, the complications of this for the optimal attack are mitigated. The only possible true result from the oracle occurs when the cookie is correct and the un-truncated is 1. Additionally, this has no effect on the greedy attack. The correct guess for malleating the leftmost bits will simply be to exclusive-OR them with zeroes. Finally, this occurs when the cookie values is all zeros, which occurs with probability $2^{-(k+1)}$, so its effect on average-case analysis is minimal.

Another possibility of overlap is that the message is already validly padded and the first $k+1$ bits are not already the cookie value. This occurs with probability slightly greater than 2^{-k}. The optimal attack is unaffected due to the truncation strategy. The greedy attack is somewhat affected in that malleating the last k bits will result in a valid padding in up to two ways: leaving the bits unchanged, and whatever transforms the bits to a k-bit representation of 1. Thus, the greedy attack in some cases gains less information when querying on an exclusive-OR which leaves padding bits unchanged. Not only does this mean that the greedy attack will de-prioritize such malleations, it also means that in this somewhat rare case the greedy algorithm is at least as inefficient as in the common case.

Alternatively, The first $k+1$ bits of the message may be equal to the cookie value. With a uniform message and a $k+1$-bit cookie, this occurs with probability $2^{-(k+1)}$. The optimal attack will be unaffected; the correct guess will not malleate these bits. However, the greedy attack is potentially affected. An oracle query on an incorrect guess attempting to malleate the last k bits into may return true if the parity of the message bits is 1, even if the last k bits were not transformed to correct padding. This case degrades the expected profitability of the greedy attack somewhat (upper-bounded by $(2^{L-(k+1)})(2^{-(k+1)})$), the class of messages which cannot entirely be ruled out times the probability of this edge case), but as k grows with some fixed l this bound shrinks proportionately to the overall profitability.

Finally, if a combination of these events occur, the complications are less clear. However, while events are not entirely independent, the probability of coincidence is roughly $2^{-((k+1)k)}$, which does not significantly affect average-case analysis for at least some k.

2.7 Realizing Plaintext Weight Functions

As discussed in §2.3, our basic attack algorithm is designed to identify the *largest subset* of messages to be eliminated at the i^{th} query. From an optimization perspective, this strategy embeds a critical assumption: namely, that every valid plaintext admitted by G_i is equally likely, and thus the most efficient path to finding M^* is to simply eliminate as many messages as possible, without regard to message "quality."

This approach can produce counterintuitive results in practice. One example manifests in our attacks on the PKCS #7 encryption padding scheme [Kal98], which we discuss in §2.10. In this format, messages may be padded with up to P bytes of padding. If our target is a valid ciphertext C^* in which $\mathsf{F}(M^*) = 1$, then plaintext candidates with $P = 1$ bytes of padding will be 2^8 times as common as candidates with $P = 2$ bytes of padding, and so on. Since our attack strategy is optimized to eliminate the *largest subset* of messages first, experiments that eliminate longer messages will tend to dominate over those that eliminate short messages. A practical result is that our attack algorithm will "assume" that short padding is more likely than longer padding, and will focus on experiments that make sense based on this assumption. Such a strategy makes sense if a typical M^* is truly 2^{112} times more likely to have $P = 1$ than to have $P = 15$. However, such a distribution may not reflect the typical distribution of messages in a real system.

Approximations with plaintext weight. To address this assumption, our attack algorithm can be updated to *weight* plaintext candidates according to some formula provided by the user. This information can be encoded via an abstract *plaintext weighting function* $W : \mathcal{M} \to \{0,\ldots,B\}$ defined with respect to some constant B, where $W(M) \to A$ defines the fractional weight $\frac{A}{B}$ of a given plaintext M. A custom weighting function can be developed, for example, to encode the assumption that all PKCS #7 padding lengths are equally likely. Returning to our padding example, a user might design W to ensure that candidates of every possible padding length are equally valid.

To apply this weighting information to the attack, we can sample a fresh universal hash function $H : \{0,1\}^m \to \{1,\ldots,B\}$ for each trial, and extend the solver query with the following additional constraint term constructed over M_0, M_1:

$$H(M_0) \leq W(M_0) \ \wedge \ H(M_1) \leq W(M_1)$$

This formulation allows the weighting function to arbitrarily reduce the number of satisfying solutions for any possible class of messages. It is easy to see that our basic attack algorithm is equivalent to using a trivial weighting function where $\forall M \in \mathcal{M}$ it holds that $W(M) = B$.

2.8 Prototype Implementation

We now describe our prototype implementation, which we call `Delphinium`. We designed `Delphinium` as an extensible toolkit that can be used by practitioners to evaluate and exploit real format oracles.

2.8.1 Architecture Overview

Figure 2.3 illustrates the architecture of `Delphinium`. The software comprises several components:

Attack orchestrator. This central component is responsible for executing the core algorithms of the attack, keeping state, and initiating queries to both the decryption oracle and SMT/SAT solver. It takes the target ciphertext C^* and a description of the functions F and $\mathsf{Maul}_{\mathsf{plain}}$ as well as the attack parameters t, δ as input, and outputs the recovered plaintext.

SMT/SAT solver. Our implementation supports multiple solver frameworks (e.g. STP [Gan18] and Z3 [MB08]) via a custom compatibility layer that we developed for our tool. To improve performance, the orchestrator may launch multiple parallel instances of these solvers.

In addition to these core components, the system incorporates two user-supplied modules, which can be customized for a specific target:

Ciphertext malleator. This module provides a working implementation of the malleation function $\mathsf{Maul}^{\Pi}_{\mathsf{ciph}}$. We realize this module as a Python program, but it can be implemented as any executable compatible with the expected interface. The interface requires input of a ciphertext and a malleation string, with output the mauled ciphertext.

Target interface (shim). This module is responsible for formatting and transmitting decryption queries to the target system. It is designed as a user-supplied module in recognition of the fact that this portion will need to be customized for specific target systems and communication channels.

As part of our prototype implementation, we provide working examples for each of these modules, as well as a test harness to evaluate attacks locally.

2.8.2 Implementation Details

Realizing our algorithms in a practical tool required us to solve a number of challenging engineering problems and to navigate limitations of existing SAT/SMT solvers.

Test Harness. For our experiments in §2.10 we developed a test harness to implement the Ciphertext Malleator and Target Interface shim. This test harness implements the code for mauling and decrypting M^* locally using a given malleation string $\mathbf{S}$.

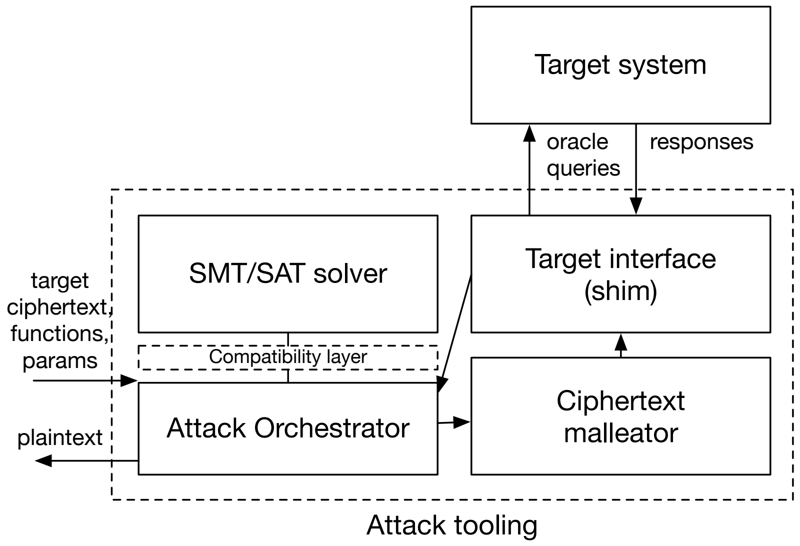


Figure 2.3: Architecture of `Delphinium`.

Low-density parity constraints. Our implementation of model counting requires our tool to incorporate $2t$ s-bit distinct parity functions into each solver query. Each parity constraint comprises an average of $\frac{n}{2}$ exclusive-ORs (where n is the maximum length of M^*), resulting in a complexity increase of tens to hundreds of gates in our SAT queries. To address this, we adopted an approach used by several previous model counting works [Zha+16; Erm+14]: using low-density parity functions. Each such function of these samples k random bits of the input string, with k centered around $log_2(n)$. As a further optimization, we periodically evaluate the current constraint formula G_i to determine if any bit of the plaintext has been fixed. We omit fixed bits from the input to the parity functions, and reduce both n and k accordingly.

Implementing `AdjustSize`. Because SAT/SMT queries are computationally expensive, our algorithms can benefit from an optimization strategy that minimizes the number of sequential queries required to maximize the value s.

One optimization is to use a logarithmic binary search procedure to increase (resp. decrease) s. This is intuitive because the other problems should be obviously satisfiable or obviously unsatisifiable at the extremes. Thus, if execution times peak around the cutoff, linear search will always execute half of the slowest queries across values of s. A classic solution to this problem is to employ binary search, skipping over many values of s which can be ruled out from being the cutoff. Unfortunately, our experiments show that the runtime efficiency gain from binary search is not clearly logarithmic, due to the fact that queries which are closer to the "actual" set size require the most solver time, and binary search spends many queries close to the target size. Moreover, this search must be carefully tuned to deal with the possibility of false negatives that can occur as a result of the probabilistic model counting procedure.

Describing `F`. In order for SMT solvers such as Z3 to reason about format checking functions using a theory of quantifier-free bitvectors, the function must be encoded as operations which are supported by this solver theory. We considered implementations comprising of bitwise operations, boolean operations, and ternary conditionals. We use Z3's interface, which allows our to process tool function descriptions encoded as Python and SMT-LIB. We also developed experimental tooling to support standard boolean circuit formats.

Describing malleation. To avoid requiring users to re-invent basic functionality, `Delphinium` provides built-in support for several malleation functions. These include simple stream ciphers, stream ciphers that support truncation (from either the left or the right side), and CBC mode encryption. The design of these malleation functions required substantial extensions to the `Delphinium` framework.

2.8.2.1 Implementing Malleation Functions

Truncation. Support for truncation requires `Delphinium` to support plaintexts of variable length. This functionality is not natively provided by the bitvector interfaces used in most solvers. We therefore modify the solver values to encode message length in addition to content. This necessitates changes to the interface for F. We accomplish this by treating the first $log_2(n)$ bits of each bitvector as a *length field* specifying how long the message is and by having every implementation of F decode this value prior to evaluating the plaintext. To properly capture truncation off either end of a message, the malleation bitvector is extended by $2\dot{l}og_2(n)$ so the lowest order $log_2(n)$ bits of the malleation bitvector specify how many bits should be truncated off the low order bits of the plaintext and the next $log_2(n)$ bits specify what should be truncated from high order bits of the message. For ease of implementation, in some schemes the n bits following the truncation describe the length field of the plaintext. This allows for easily expressing the exclusive-OR portion of our malleation without bit-shifting and allows encoding extension. Some schemes, such as stream ciphers, only enable truncation off one side of the message, and so in this case we add a constraint to the formula which disallows truncation off the low order bits of a message. This is because trunction off the high order bits would imply a misalignment of the ciphertext with the keystream, causing decryption to produce effectively randomized plaintext.

Truncation for Block Cipher Modes. In block cipher modes such as CTR, CFB, and OFB, an attacker also has the ability to increment the nonce and truncate off blocks of ciphertext. This is not necessarily possible when dealing with other stream ciphers, due to the keystream being misaligned with the ciphertext. To capture this capability, in the malleation function we additionally constrain the malleation string to express truncation off the high order bits of a message (earlier blocks of ciphertext), provided the number of bits being truncated is a multiple of the block size.

CBC Mode. In contrast with stream ciphers, $\mathsf{Maul}_{\mathsf{plain}}^{\Pi_{\mathsf{CBC}}}$ is not equal to $\mathsf{Maul}_{\mathsf{ciph}}^{\Pi_{\mathsf{CBC}}}$ and moreover $\mathsf{Maul}_{\mathsf{plain}}^{\Pi_{\mathsf{CBC}}}$ is significantly more complex. In CBC mode, decryption of a ciphertext block C_i is defined as $P_i = Dec_k(C_i) \oplus C_{i-1}$ where C_{i-1} denotes the previous ciphertext block. Since the block C_i is given directly to a block cipher, any implementation must account for the the fact that modification of the block C_i creates an unpredictable effect on the output P_i, effectively randomizing it via the block cipher.

For a solver to reason over such an effect on the plaintext output, we would need to include constraint clauses corresponding to encryption and decryption, i.e. boolean operations implementing symmetric schemes like AES. To avoid this significant overhead, we instead modify the interface of $\mathsf{Maul}_{\mathsf{plain}}^{\Pi_{\mathsf{CBC}}}$ to output two abstract bitvectors (M, Mask). Mask represents a *mask string*: any bit j where $\mathsf{Mask}[j] = 1$ is viewed as a *wildcard* in the message vector M. When $\mathsf{Mask}[j] = 0$, the value of the output message is equal to $M[j]$ at that position, and when $\mathsf{Mask}[i] = 1$ the value at position $M[j]$ must be viewed as unconstrained. This requires that we modify F to take (M, Mask) as input. The modified F is able to produce a third value in addition to true and false. This new output value indicates that the format check cannot assign a definite true/false value on this input, due to the uncertainty created by the unconstrained bits. In practice, we implement the output of F as a bitvector of length 2, and modify our algorithms to use 00 and 01 in place of 0 and 1, respectively. Realizing this formulation requires only minor implementation changes to our core algorithms.

Exclusive-OR and Truncation for CBC. With CBC mode decryption, manipulating a preceding ciphertext block C_{i-1} produces a predictable exclusive-OR in the plaintext block P_i. A message that has been encrypted with a block cipher can also be truncated, provided that truncation is done in multiples of the block size. Therefore, we define malleability for CBC to capture (1) block-wise truncation (from either the left or right side of the ciphertext) and (2) exclusive-OR, where exclusive-OR at index i in one block produces the corresponding bit-flip at index i in the next block of decrypted ciphertext.

Supporting Extension. For encryption schemes that allow truncation off the beginning of a message, an attacker may also be able to fill in the truncated portion with arbitrary ciphertext, even if this ciphertext may decrypt to plaintext unknown to them. If the corresponding portion of the plaintext is not examined by the format check function, the attacker can derive information from such queries (if the portion is checked, the attacker can only learn the result of the check over random bits by nature of ciphers). Thus, we create an additional initial constraint for this special case, which allows extension to the ciphertext, limited to where the corresponding plaintext is not examined by the format function.

2.8.3 Further Implementation Considerations

Here we describe a number of challenges we encounter pursuing practicality in terms of computational efficiency. We primarily consider efficiency to be measured as wall-clock time to a solution as we expect this to be the primary relevant *practical* (rather than theoretical) concern to a non-expert user of our system. We provide our approaches to solving or mitigating the impact of these challenges.

2.8.3.1 Selecting SAT and SMT solvers

Our initial work explored using Z3 [MB08] due to its robust feature set and continued support. We also attempted to use other SMT solvers including Boolector [Nie+18], which is optimized for bitvector operations, and STP [Gan18], which supports CryptoMiniSAT (CMS) as a backend SAT solver. CMS [SNC09] is specifically optimized for bitvector operations involving exclusive-ORs, which in standard CNF are represented by expressions of exponential size in the number of variables. CMS is able to recover and simplify exclusive-ORs in order to make their processing far more efficient in terms of concrete runtime, and additionally allows an extension of the CNF format which includes exclusive-ORs represented in expressions only linearly sized in the number of variables.

Unfortunately we found Boolector (via the Python bindings) unusable due to software issues, and STP unusable due to significant overhead in constraint processing. As such, we employ a hybrid architecture which exploits the benefits of both Z3 and CMS while avoiding most of the aforementioned issues. In the course of developing `Delphinium` we discovered and either fixed or worked with maintainers to fix various software issues in these libraries. Fixing these issues was a necessary step, albeit one which hindered our software development significantly and impacted our ability to run end-to-end tests.

Combining Z3 and CMS via a CNF bridge. To improve performance, we adopted CMS, which has been used extensively in the model counting literature [SNC09] due to its ability to simplify exclusive-OR clauses. CMS is natively supported by the STP SMT solver, which provides powerful constraint simplification routines. Unfortunately, our experiments with STP exposed a number of other performance limitations, as well as intermittent bugs in formulae with large (> 64 bit) bitvectors. As an optimization, we adopted a hybrid model that uses Z3 to periodically simplify the constraint system and generate partial CNF files for CMS to reason over; we subsequently edit these files directly to add exclusive-OR gates for model counting, and to include new constraints derived from decryption oracle queries. We note that this approach is suboptimal from a performance perspective, and that there is still room for performance improvement. Nonetheless, we found that it systematically outperformed the tested alternatives.

CNF manipulation. In order to aid development of `Delphinium`, we developed tools for manipulating both textual CNF files and bespoke Python CNF objects. We include these tools in our contribution for general use, as the DIMACS CNF format is widely used in SAT/SMT work. These tools include efficiently adding exclusive-OR gates to the "CNFXOR" files which CMS supports, and adding translation layers which allow for efficient recovery of DIMACS SAT models for programs interfacing with Z3.

2.8.3.2 Employing parallelization

Modern SMT and underlying SAT solving strategies are amenable to parallelization. Additionally, as a result of our optimizing search for a maximum satisfiable problem, our algorithm is also highly parallelizable – a set of satisfiability checks can be run with MBound constraints which require subsets of various sizes can be run in parallel, and the result from the largest satisfiable size taken. We refer to the width of the range of sizes run in parallel as the *window size* for parallelization. In order to effectively allocate cores, we must balance increased window sizes with additional cores per solver instance. Anecdotally, it is faster to run larger windows with fewer cores per solver – an unsurprising indication that modern solvers do not use multiple cores at 100% utilization.

Efficiently replicating a solver instance is a challenge across SMT implementations (Z3 and STP) which led to multiple redesigns of our implementation. This is mostly caused by features of the underlying C++ implementations which employ static variables, to which pointers are abstractly operated over in the Python bindings we use. In order to circumvent these challenges while avoiding the overhead of e.g. the *fork* system call we use a hybrid implementation which involves a Z3 solver instance in a single process which exports its representation of the constraints as CNF, which we can easily replicate and pass to many parallel instances of CMS after adding the appropriate exclusive-OR constraints. By exporting CNF before adding XORs we avoid slow CNF generation due to exponentially-sized XOR representations.

2.8.3.3 Tseitin conversion and CNFXOR

In order to enable parallelization and to avoid the runtime overhead introduced by STP, we export constraint systems to conjunctive-normal form (CNF) before solving via CMS. This allows a set of constraints to be rapidly replicated for use by many independent processes. However, XOR constraints such as those required by our hash functions expand to CNF which is exponentially-sized in the number of variables in the XOR, and as such we experienced massive runtime overhead in Z3 when exporting our model counting constraint systems. In order to work around this, we export CNF excluding the XORs required by the model counting algorithm, and then append these XORs to the CNF output. CMS is able to process such XOR-supplemented CNF (CNFXOR) files efficiently, and they represent XORs in linearly- rather than exponentially-sized output.

2.8.4 Software

Our prototype implementation of `Delphinium` comprises roughly 4.2 kLOC of Python. This includes the attack orchestrator, example format check implementations, the test harness, and our generic solver Python API which allows for modular swapping of backing SMT solvers, with implementations for Z3 and STP provided. In pursuing this prototype, we submitted various patches to the underlying theory solvers that have since been included in the upstream software projects.

Also included are functions which allow instantiations of Gomes *et al.*'s [GSS06] model counting tests with configurable parameters, CNF generation using the Z3 Tactic API and strategies for recovering solutions from CMS, CNFXOR generation for use in CMS, and translation from CMBC-GC boolean circuit output (compilations of ANSI C programs to boolean circuits) to Python for use with the Python solver API.

2.8.5 Extensions

In general, arbitrary functions on fixed-size values can be converted into boolean circuits which SMT solvers can reason over. Existing work in MPC develops compilers from DSLs or a subset of C to boolean circuits which could be used to input arbitrary check format functions easily [Moo+16; Fra+14]. Experimenting with these, we find that the circuit representations are very large and thus have high runtime overhead when used as constraints. It is possible that circuit synthesis algorithms designed to decrease circuit size (used for applications such as FPGA synthesis) or other logic optimizers could reduce circuit complexity, but we leave exploring this to future work.

For complex format checking functions where only compiled implementations are available, existing works in function approximation (used commonly in fuzzing) can be leveraged to extract approximations of format check functions, against which possible candidate attacks could be evaluated. Heuristically derived C function approximations are developed in the American Fuzzy Lop fuzzer and in NeuEx, a machine-learning driven fuzzing system [Zal; She+18]. We additionally provide a translation tool from the output format of CMBC-GC [Fra+14] to Python (entirely comprised of circuit operations) to enable use of the Python front-end to `Delphinium`. We leave the empirical evaluation of such heuristic approximation techniques to future work. Emerging techniques in symbolic execution such as those explored in [Wan+19] may provide an additional avenue for format function constraint generation in future work.

2.9 Format Check Implementations

We present a listing of several Python-Solver format check functions below to elucidate our implementation and to illustrate the relative simplicity of employing our tool rather than manually evaluating various format check functions for their inherent information leakage.

2.9.1 PKCS #7 padding

```
def checkFormatPKCS7(padded_msg, solver):
    """ PKCS7 check format as solver constraints """
    # size of a byte
    unit_size = 8
    # number of units which make up a block
    block_size = 16
    # base case of an OR iteratively constructed
    is_valid = solver.false()
    # for each possible padding size
    for i in range(1, block_size+1):
        # base case of an AND iteratively constructed
        correct_pad = solver.true()
        # for each byte of a pad of size i
        for j in range(i):
            # extract from padded_msg the bits
            # which should make up a padding byte
            pad_byte = solver.extract(
                            padded_msg,
                            (j + 1)*unit_size-1,
                            j*unit_size)
            # pad is correct if this and all previous
            # checked bytes match the bitvector
            # representing the size
            correct_pad = solver._and(
                            correct_pad,
                            solver._eq(
                                solver.bvconst(i, unit_size),
                                pad_byte))
        # padding is valid if one size matched
        # thus, return the OR of padding checks at each size
        is_valid = solver._or(is_valid, correct_pad)
    return is_valid
```

2.9.2 AWS Session Ticket

```
def checkFormatAWSSessionTicket(full_msg, time, state, mall=0, trunc=0):
    # If value is changed for these fields, will fail
    if grabByte(mall, 1) != 0:
        return 2
    if grabNBytes(mall, 2, 2) != 0:
        return 2

    #  message must be at least S2N_STATE_SIZE_IN_BYTES bytes long
    if full_msg & ((1 << length_field_size)-1) != test_length:
        return 0
    msg = full_msg >> length_field_size
    # first byte must match S2N_SERIALIZED_FORMAT_VERSION
    if grabByte(msg,0) != S2N_SERIALIZED_FORMAT_VERSION:
        return 0

    # protocol version from earlier point in time
    # this is stateful information
    if grabByte(msg, 1) != state["proto_version"]:
        return 0

    # iana value of cipher suite negotiated, this is also stateful information
    if grabNBytes(msg, 2, 2) != state["iana"]:
        return 0

    # checking expiry of the session ticket, this is also stateful
    time_on_ticket = grabNBytes(msg, 8, 4)
    # this is going to change every time it's called...
    if time_on_ticket > time:
        return 0
    if (time - time_on_ticket) > TICKET_LIFETIME:
        return 0

    return 1
```

2.9.3 Bitwise padding

```
def checkFormatBitwisePadding(padded_msg, solver):
    numBits = solver.extract(padded_msg, paddingSizeBits-1, 0)
    compound_expr = solver.true()
    numVal = 1
    for i in range(1, ciphertextBits-paddingSizeBits+1):
        ones = solver.extract(padded_msg, paddingSizeBits+i-1, paddingSizeBits)
        compound_expr = solver._if(solver._eq(numBits, i),
                                   solver._eq(ones, solver.bvconst(numVal,i)),
                                   compound_expr)
        numVal = (numVal << 1) | 1
    length_check = solver._ule(numBits,ciphertextBits-paddingSizeBits)
    return solver._and(length_check, compound_expr)
```

2.10 Experiments

In this section we evaluate `Delphinium` against a variety of encryption and format checking functionalities. We include examples ranging from demonstrative to practical, re-discovering known attacks and generating novel ones. We further include a negative result as well as examples which stretch the system to its limits in order to extrapolate feasibility against further, unknown, or more complex targets.

2.10.1 Experimental Setup

To evaluate the performance of `Delphinium`, we tested our implementation on several multi-core servers using the most up-to-date builds of Z3 (4.8.4) and CryptoMiniSAT (5.6.8). The bulk of our testing was conducted using Amazon EC2, using compute-optimized `c5d.18xlarge` instances with 72 virtual cores, 144GB of RAM, and 900GB of extraneous ephemeral EC2 storage. Several additional tests were run a 72-core Intel Xeon E5 CPU with 500GB of memory running on Ubuntu 16.04, and a 96-core Intel Xeon E7 CPU with 1TB of memory running Ubuntu 18.04. We refer to these machines as AWS, E5 and E7 in the sections below.

Data collection. For each experimental run, we collected statistics including the total number of decryption oracle queries performed; the wall-clock time required to construct each query; the number of plaintext bits recovered following each query; and the value of s used to construct a given malleation string. We also recorded each malleation string $\mathbf{S}$ produced by our attack, which allows us to "replay" any transcript after the fact. The total number of queries required to complete an attack provides the clearest signal of attack progress, and we use that as the primary metric for evaluation. However, in some cases we evaluate partial attacks using the `ApproxMC` approximate model counting tool [SM19]. This tool provides us with an estimate for the total number of remaining candidates for M^* at every phase of a given attack, and thus allows evaluation of partial attack transcripts.

Selecting attack parameters. The adjustable parameters in `FastSample` include t, the number of counting trials, δ, which determines the fraction of trials that must succeed, and the length of the parity constraints used to sample. We ran a number of experiments to determine optimal values for these parameters across the format functions PKCS7 and a bitwise format function defined in §2.10.2. Empirically, $\delta = 0.5$,

$2 \leqslant t \leqslant 5$, and parity functions of logarthmic length are suitable for our purposes. These tests were performed on AWS.

2.10.2 Experiments with Stream Ciphers

Because the malleation function for stream ciphers is relatively simple (consisting simply of bitwise exclusive-OR), we initiated our experiments with these ciphers.

Byte-wise Encryption Padding. The PKCS #7 encryption standard (codified in RFC 2315) [Kal98] defines a padding scheme for use with block cipher modes of operation. This padding is similar to the standard TLS CBC-mode padding [AP13] considered by Vaudenay [Vau02]. We evaluate our algorithm on both these functions as a benchmark because PKCS7 and its variants are reasonably complex, and because the human-developed attack is well understood. We refer to these schemes as PKCS7 and TLS-PKCS7, respectively.

The PKCS7 function operates by padding an octet-aligned message to a multiple of B bytes, where B is typically a cipher's block size. The padding string is of length $1 \leq P \leq B$ bytes, and each byte comprises an 8-bit unsigned encoding of P. The format checking function $\mathsf{F}_{\mathsf{PKCS7}}$ recovers P from the final byte of the padded message, verifies that $P \in \{1,\ldots,B\}$, and checks that each of the trailing P bytes contains the same value. We present a Z3-Python implementation in §2.9. TLS-PKCS7 is nearly identical, with only small differences: first, the byte used to pad is not the value P, but rather $P-1$, and second, padding may extend beyond the last block of the message. Neither of these differences complicates the constraint formula significantly.

Setup. We conducted an experimental evaluation of the PKCS #7 attack against a 128-bit stream cipher, using parameters $t = 5, \delta = 0.5$. Our experiments begin by sampling a random message M^* from the space of all possible PKCS #7 padded messages, and setting $G_0 \leftarrow \mathsf{F}_{\mathsf{PKCS7}}$. This plaintext distribution tends to produce messages with short padding. This evaluation was performed on AWS, E5, and E7. We also implemented the manually-developed "human" attack (based on Vaudenay's approach [Vau02]) as a baseline for comparison.

Results. Our four complete attacks completed in an average of 1699.25 queries (min. 1475, max. 1994) requiring 1.875 hours each (min. 1.63, max. 2.18). A visualization of one such attack is shown in §2.11.1. These results compare favorably to the Vaudenay attack, which requires ~2000 queries in expectation. However it is likely that additional tests would find some examples in excess of this average. As points of comparison, attacks with $t = 3$ resulted in a similar number of queries (modulo expected variability over different randomly sampled messages) but took roughly 2 to 3 times as long to complete, and attacks with $t = 1$ reached over 5000 queries having only discovered half of the target plaintext message.

Bitwise Padding. To test our attacks, we constructed a simplified *bit* padding scheme $\mathsf{F}_{\mathsf{bitpad}}$. This contrived scheme encodes the bit length of the padding P into the rightmost $\lceil log_2(n) \rceil$ bits of the plaintext string, and then places up to P padding bits directly to the left of this length field, with each padding bit set to 1. We verified the effectiveness of our attacks against this format using a simple stream cipher. Using the parameters $t = 5$, $\delta = 0.5$ the generated attacks took on average 153 queries (min. 137, max. 178). Figure 2.1 shows one attack transcript at $t = 5, \delta = 0.5$. These experiments were run primarily on E5.

Negative result: Cyclic Redundancy Checks (CRCs). Cyclic redundancy checks (CRCs) are used in many network protocols for error detection and correction. CRCs are well known to be malleable, due to the linearity of the functions: namely, for a CRC it is always the case that $\mathrm{CRC}(a \oplus b) = \mathrm{CRC}(a) \oplus \mathrm{CRC}(b)$.

While the basic algorithm is intuitive, there are numerous variations of CRCs in existence. Each CRC-n is defined by the degree n of the generator polynomial, the generator's coefficients, the order that the input data will be processed in and the output order (which is sometimes called input/output reflection), the input mask, and the output mask. The calculation for equality also differs in some implementations: a CRC computation is done over the data stream concatenated with a number of 0's equivalent to the CRC field size and the resulting equality check is done by computing a CRC on the original data stream concatenated with the CRC and checking if the remainder is 0.

To test `Delphinium`'s ability to *rule out* attacks against format functions, we implemented a message format consisting of up to three bytes of message, followed by a CRC-8 and a 5-bit message length field. The format function $\mathsf{F}_{\mathsf{crc8}}$ computes the CRC over the message bytes, and verifies that the CRC in the message matches the computed CRC. We used a simplified implementation that does not reflect input an output, or add an initial constant value before or after the remainder is calculated. These simplifications do not meaningfully affect the outcome of our experiments for the purposes of confirming a negative result.

A key feature of this format is that a valid ciphertext C^* should *not* be vulnerable to a format oracle attack using a simple exclusive-OR malleation against this format, for the simple reason that the attacker can predict the output of the decryption oracle for every possible malleation of the ciphertext (due to the linearity of CRC), and thus no information will be learned from executing a query. This intuition was confirmed by our attack algorithm, which immediately reported that no malleation strings could be found. These experiments were performed on $\mathsf{E5}$.

2.10.3 Ciphers with Truncation

A more powerful malleation capability grants the attacker to arbitrarily *truncate* plaintexts. In some ciphers, this truncation can be conducted from the low-order bits of the plaintext, simply by removing them from the right side of the ciphertext. In other ciphers, such as CTR-mode or CBC-mode, a more limited left-side truncation can be implemented by modifying the IV of a ciphertext. `Delphinium` includes malleation functions that incorporate all three functionalities.

CRC-8 with a truncatable stream cipher. To evaluate how truncation affects the ability of `Delphinium` to find attacks, we conducted a second attack using the function $\mathsf{F}_{\mathsf{crc8}}$, this time using an implementation of AES-CTR supporting truncation. Such a scheme may seem contrived, since it involves an encrypted CRC value. However, this very flaw was utilized by Beck and Tews to break WPA [TB09]. In our experiment, the attack algorithm was able to recover two bytes of the three-byte message, by using the practical strategy of truncating the message and iterating through all possible values of the remaining byte. These experiments were run primarily on `E5`.

Using truncation, differentiation can be accomplished for all but the last byte: consider a message encrypting a properly formatted CRC-8 message of two bytes denoted as $d_{15}d_{14}d_{13}\ldots d_0 \| c$ (with $\|$ as concatenation) where c denotes the CRC value and $d_{15}\ldots d_0$ the bits of the message. If we truncate the message to be $d_7 \ldots d_0 \| c$, c is no longer valid. But precisely how has it been invalidated? Considering our message as a polynomial and the CRC generator polynomial as $b(x)$, by the division algorithm there must exist $q(x)$, $r(x)$ such that

$$r(x) = \sum_{i=0}^{15} d_i \cdot x^i + q(x)b(x)$$

The correct CRC for our truncated message $\sum_{i=0}^{7} d_i \cdot x^i$ can then be calculated as

$$(r(x) + \sum_{i=8}^{15} d_i \cdot x^i) \pmod{b(x)} = \sum_{i=0}^{7} d_i \cdot x^i + q(x)b(x)$$

Therefore, if we want to test whether a ciphertext message begins with $d'_{15} \ldots d'_8$ we can truncate off the end of the ciphertext and send the modified CRC: $r(x) \oplus (\sum_{i=8}^{15} d'_i * x_i)$ (mod $b(x)$). If our guess for the plaintext was correct, the new CRC is correct for the truncated message, since $\text{CRC}(d_{15} \ldots d_0) \oplus \text{CRC}(d'_{15} ... d'_8 \| 0 \ldots 0\}) = \text{CRC}(0 \ldots 0 \| d_7 \ldots d_0)$. A human attack could then learn bytes starting at the the end of the message by truncating and modifying the CRC in each query.

As this example demonstrates, the level of customization and variation in how software developers operate over encrypted data streams can obfuscate the concrete security of an existing implementation. This illustrates the utility of `Delphinium` since such variation's effect on the underlying scheme does not need to be fully understood by a user, outside of encoding the format's basic operation.

Thumb Embedded ISA. To exercise `Delphinium` against a novel format oracle of notably different structure than those traditionally analyzed (such as padding), we implemented a minimal instruction interpreter for the 16-bit Thumb instruction set architecture (ISA), defined as part of the ARM specification [Lim18], capable of emitting illegal instruction signals. Then, operating over stream-cipher encrypted Thumb instructions and using illegal instructions as a boolean signal, `Delphinium` is able to exploit the exclusive-OR malleation to uncover the top seven bits of each 16-bit instruction, in many cases uncovering nine or more (up to 16) bits of each instruction, in an average of ~13.3 queries, with each full attack taking only seconds on `E5`. Such a partial firmware decryption generally leaks the instruction opcode, but not its arguments. This could be very useful to an attacker, for example in fuzzy comparison with compiled open source libraries to determine libraries and their versions used in a given firmware update.

Although limited in a few regards, most notably in the simplification of the format oracle into a boolean signal and the assumption that an attacker could be situated in a way that this signal could be gathered, this attack is timely in that it is inspired by the widespread use of unauthenticated encryption in device firmware updates[Ecl20]. If these updates are delivered over-the-air, they may be susceptible to man-in-the-middle attacks enabling such a decryption oracle. Extensive industry research and a current Internet Draft note that unauthenticated firmware updates are an ongoing problem[Ecl20; MTB20].

This initial result serves both as validation of `Delphinium` and as creation of an avenue for future work, including the development of a model for a more complex but widespread ISA such as 32-bit ARM[Lim18], perhaps exploiting additional signals such as segmentation faults or side channels in order to capture the capabilities of a sophisticated adversary.

S2N with Exclusive-OR and Truncation. S2N is an open source TLS library that was created by Amazon Web Services [Sch15]. One of the features includes an implementation of the recently standardized TLS 1.3. Part of the protocol for TLS 1.3 includes a specification for session resumption, which is used when a client and server have an already established connection but would like to start up a new session without re-authenticating one another. The client receives a session ticket from the server that it can use to resume state at a later date. The S2N library delivers a concrete implementation of this protocol in the form of a 60 bytes session state value with the following format (each field concatenated):

$$[RANDOMSECRET][TIMEFIELD][CIPHERSUITE]$$

$$[PROTOCOLVERSION][FORMATVERSION]$$

The 48-byte $RANDOMSECRET$ denotes a pre-master secret that is used as keying material, and the 8-byte $TIMEFIELD$ denotes when the ticket was issued. After authentication and decryption of the AES-GCM encrypted ticket, various checks are performed on values embedded in the ticket. Depending upon these values, a server either continues communicating with an endpoint or may error. In other words, we may interpret the check of the ticket as a format and feed in a description of it to `Delphinium`. While some may question the efficacy of such an attack, given that the session ticket is encrypted with an AEAD cipher, it is possible for such an attack to proceed provided that the authentication component of the cipher is broken. Indeed, it has been shown in the past that reusing a nonce in AES-GCM with small tags results in a higher than expected chance of forgery and a series of successful forgery attempts even leads to recovery of the authentication key [Fer05]. Nonce reuse is also not that rare of an occurrence and has been seen in the wild in at least one internet wide scan [Böc+16]. To evaluate a realistic attack on a practical format function, we developed a format checking function for the Amazon `s2n` [Sch15] TLS session ticket format. `s2n` uses 60-byte tickets with a 12-byte header comprising a protocol version, ciphersuite version, and format version, along with an 8-byte timestamp that is compared against the current server clock. Although `s2n` uses authenticated encryption (AES-GCM), we consider a hypothetical scenario where nonce re-use has allowed for message forgery [Fer05; Böc+16].

Our experiments recovered the 8-byte time field that a session ticket was issued at: in one attack run, with fewer than 50 queries. However, the attack was unable to obtain the remaining fields from the ticket. This is in part due to some portions of the message being untouched by the format function, and due to the complexity of obtaining a positive result from the oracle when many bytes are unknown. We determined that a full attack against the remaining bytes of the ticket key is possible, but would leave 16 bytes unknown and would require approximately 2^{50} queries. Unsurprisingly, `Delphinium` timed out on this attack. These experiments were run on AWS and E5.

2.10.4 CBC mode

We also used the malleation function for CBC-mode encryption. This malleation function supports an arbitrary number of blocks, and admits truncation of plaintexts from either side of the plaintext. In practice, truncation in CBC simply removes full blocks from either end of the ciphertext. The CBC malleation function accepts a structured malleation string S, which can be parsed as (S', l, r) where l, r are integers indicating the number of blocks to truncate from the message.

To test this capability, we used the PKCS7 format function with a block size of $B = 16$ bytes, and a two-block CBC plaintext. (This corresponds to a ciphertext consisting of three blocks, including the Initialization Vector.) `Delphinium` generated an attack which took 3441 oracle queries for a random message with four bytes of padding. This compares favorably to the Vaudenay attack, which requires 3588 queries in expectation. Interestingly, `Delphinium` settled on a more or less random strategy of truncation. Where a human attacker would focus on recovering the entire contents of one block before truncating and attacking the next block of plaintext, `Delphinium` instead truncates more or less as it pleases: in some queries it truncates the message and modifies the Initialization vector to attack the first block. In other queries it focuses on the second block. Figure 2.4 gives a brief snapshot of this pattern of malleations discovered by

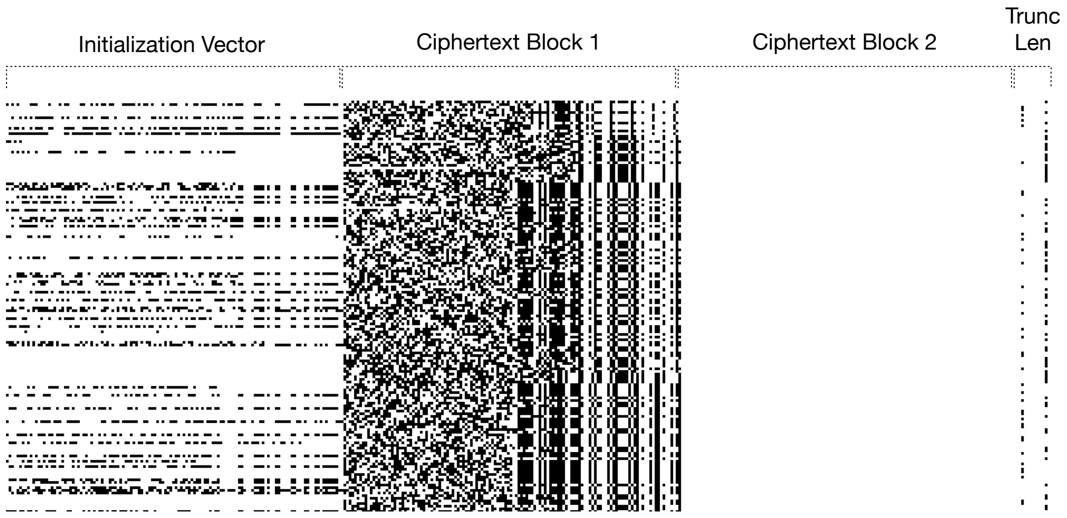


Figure 2.4: A contiguous set of malleation queries made by `Delphinium` during a simulated CBC attack. The rightmost bits signal truncation (from left or right).

`Delphinium`. Despite this query efficiency (which we seek to optimize, over wall-clock efficiency), the compute time for this attack was almost a week of computation on `E5`.

Additionally, we created a networked instantiation of the attack against TLS-PKCS7. In this implementation, malformed ciphertexts are transmitted over a local network to the decrypting endpoint, which then replies with the leaked boolean signal used to update the solver. We include this for completeness, despite its equivalence to the shim approach with respect to evaluating `Delphinium`.

Sudoku. SAT solvers have been used many times to solve Sudoku puzzles, *e.g.,* [Rie+17; Ran18; ppm18]. To evaluate our techniques against arbitrary formats, we instantiated a function $\mathsf{F}_{\mathsf{sudoku}}$ that interprets a plaintext M as an encoded $D \times D$ board, and outputs 1 if and only if the puzzle is valid.

Because existing implementations *e.g.,* [ppm18] make use of algebraic theories (which are not easily translated to our setting), we wrote a new constraint formula using quantifier-free bitvectors. Each square of the puzzle is encoded using $\lceil log_2(D+1) \rceil$ bits, with 0 corresponding to an unfilled square. Because our constraint systems become very large for standard 9×9 puzzles, we tested our system against 4×4 puzzles. In this format, each square is represented by a 3-bit substring, encoding the values $\{0, \ldots, 7\}$. This is a very simple format, since there are only 280 possible valid puzzles.

We tested this format using a simple stream cipher, beginning with 10 random (valid) Sudoku puzzles, and with $G_0 = \mathsf{F}_{\mathsf{Sudoku}}$, $t = 5$ and $\delta = 0.5$. We found that our attack was able to derive the correct plaintext Sudoku solution in an average of 20 queries, with the attacks running in an average of 33 minutes. The variation in the attack statistics is large (min. 7, max. 32 queries; min. 10, max. 60 minutes). This occurs because the algorithm is able to derive many more constraints on its model when the oracle returns a True result, and this occurs only in the relatively rare case where the chosen malleation induces a permutation on some subset of the entries, preserving the uniqueness of values in rows, columns, and squares, and without changing values to anything outside of $\{1, \ldots, 4\}$. Each attack completed after it found the second such malleation; the variation comes from invalid guesses. A visualization of one attack can be found in §2.11.5.

2.11 Additional Experimental Results

We provide this section to demonstrate further experimental evaluation performed to validate `Delphinium` against both practical and novel targets.

2.11.1 PKCS #7

Figure 2.5 demonstrates an example sequence of malleations strings displayed as bitmaps, with one malleation string per row. This image represents a full attack, constituted of 1475 queries, against PKCS #7 with a 128-bit plaintext encrypted with a stream cipher. As the attack progresses, bytes are discovered from right to left in a similar fashion to the human attack. This emergent behavior occurs as it is the most efficient way to discover the underlying plaintext bytes, in expectation.

2.11.2 Bitwise Padding Format Tests Over t

We empirically observe the parameter δ has limited impact on solver runtime and set $\delta = 0.5$., i.e. requiring all trials to be satisfied. This produces reasonable performance and is consistent with previous works [GSS06]. After running a variety of test we determined that t is a key parameter affecting attack runtime. Analysis confirmed that larger t increases query profitability by increasing the certainty of the underlying counting formula [GSS06]. However, larger t significantly increases the complexity of each constraint formula, which results in an (approximately linear) increase in CNF complexity and a variable increase in solver runtime depending on the underlying constraints. Additionally, despite increased runtime, we find that increasing t is subject to rapidly diminishing returns. As such, we generally use the lowest value for t which empirically succeeds. Figure 2.6 demonstrates malleation string bitmaps for full attacks against the bitwise padding format for $t \in 2, \ldots, 5$. The $t = 1$ bitmap is truncated significantly.

Additionally with the bitwise format, for each t from $\{1, \ldots, 6\}$ we execute a full attack, once each for logarithmically-sized and full hash functions. We use `ApproxMC` to evaluate generated queries by model counting the messages which are excluded by a given query. This value is tightly related to the remaining valid candidate messages and as such the experiments are comparable.

To evaluate the impact of changing t on our attacks against the bytewise PKCS #7 padding format, we conducted a series of "microbenchmarks" of our attack's experiment generation procedure. In each experiment, we asked the algorithm to derive 10 malleation strings $\mathbf{S}$ at arbitrary points along a complete attack run, and then used

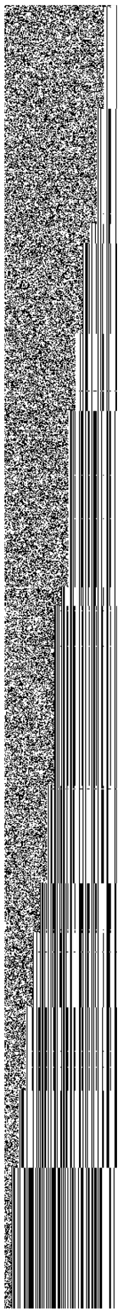


Figure 2.5: PKCS #7 malleation bitmap with characteristic “staircase” pattern as the attack progresses through the ciphertext right to left. The attack proceeds top to bottom, with one row per iteration of the attack.

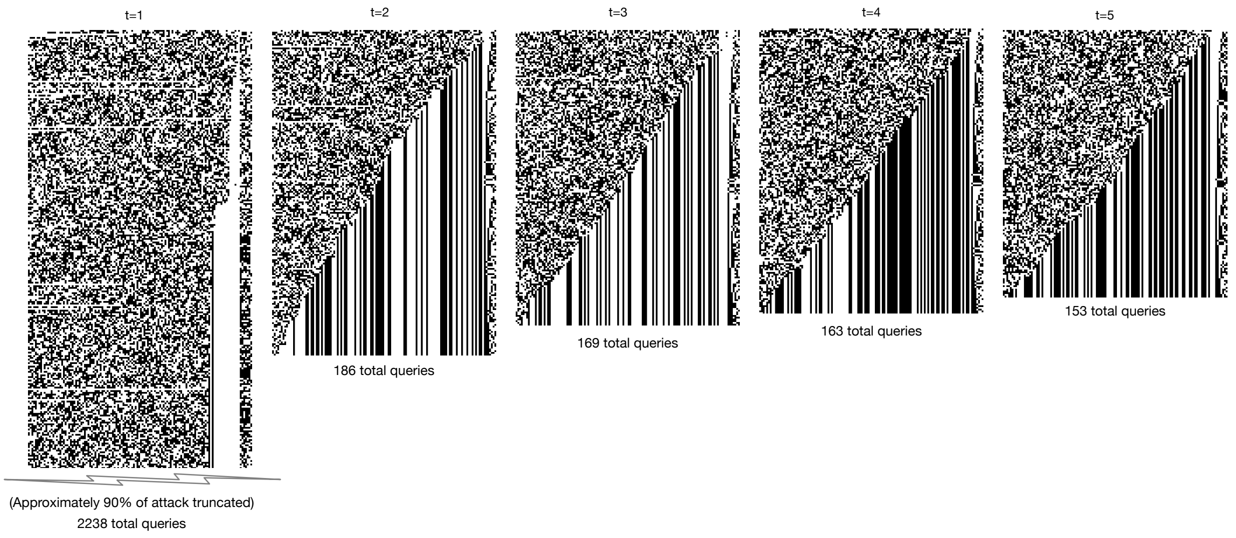


Figure 2.6: Malleation strings for 128-bit bitwise padding ($\mathsf{F}_{\mathsf{bitpad}}$) using a stream cipher. Experiments consider several different values of t, all with $\delta = 0.5$. Each uses a different random 128-bit message, which can account for some variability in the attack progress. For $t = 1$ the total number of queries is too large to show, and so we only present approximately the first 10% of the full attack transcript.

the `ApproxMC` model counting tool to measure and average the minimum profitability of the resulting malleation string.We did this by simulating each possible result from the decryption oracle, and using `ApproxMC` the number of plaintext candidates eliminated by each. We repeated each experiment for each $t \in \{1, \ldots, 6\}$, and measured the wall-clock execution time of experiment generation. This strategy is necessarily heuristic, but is a useful method for indicating appropriate parameters for a full attack run without *a priori* knowledge of attack performance.

2.11.3 Differences between Short and Long Exclusive-OR

The following figures demonstrate attacks on the 128-bit Bitwise Padding Format, highlighting differences when short (2.7) or long (2.8) exclusive-or expressions are used.

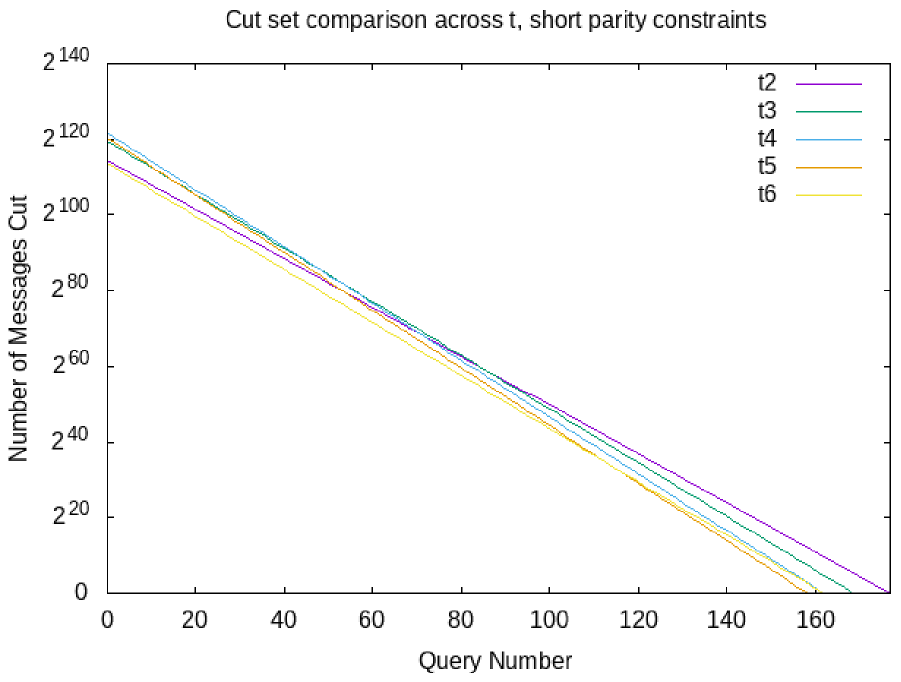


Figure 2.7: Approximate number of removed candidate messages for each query in the attack, with parity constraints of size $\log n$ (i.e. short).

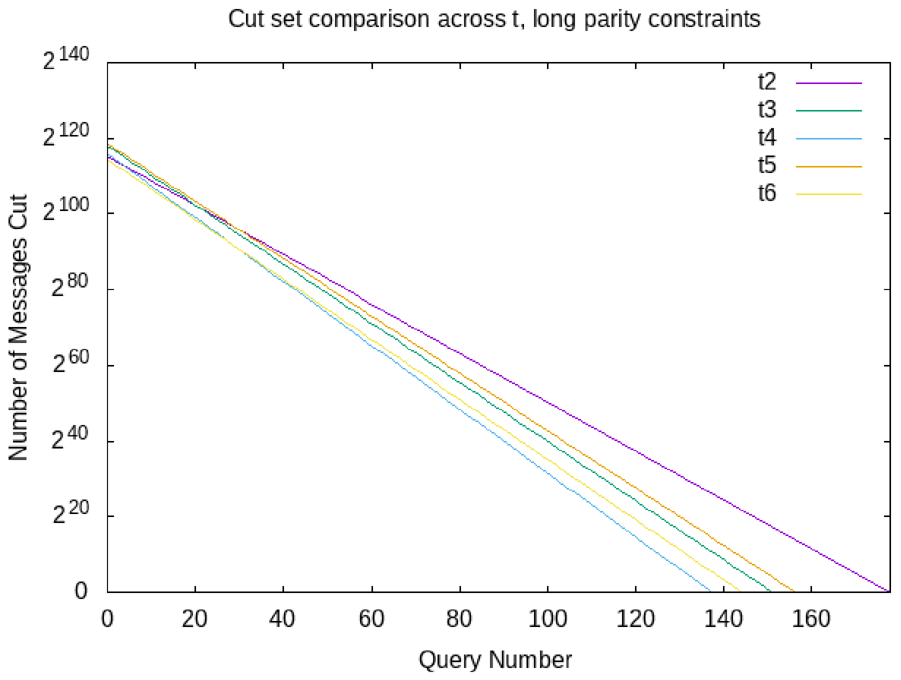


Figure 2.8: Approximate number of removed candidate messages for each query in the attack, with parity constraints of size $\frac{n}{2}$ (i.e. long).

2.11.4 CRC

Figure 2.9 displays a bitmap of malleation strings for an attack against CRC, allowing truncation and exclusive-OR malformations.

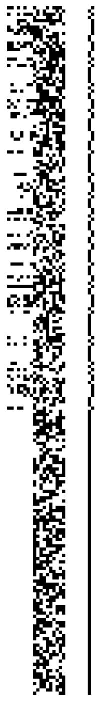


Figure 2.9: Representation of a malleation string an attack on a CRC-8 where the length of the data to compute the CRC over is 3 bytes long.

2.11.5 Sudoku

In Figure 2.10 we show an attack against the Sudoku format check. This format maps a 48-bit bitvector to entries into a 4x4 Sudoku board and checks the validity of the resulting solution. Values are coded as 3 bits, allowing entries from 0 to 7 (where 0 is coded as an unfilled square). Initially, the solver knows nothing about the real plaintext, which is a given Sudoku solution. The mauled boards show the malleation chosen by the solver applied to the real plaintext. After 7 queries, the solver has uniquely constrained its model for the real plaintext, which is correct. When the solver chooses

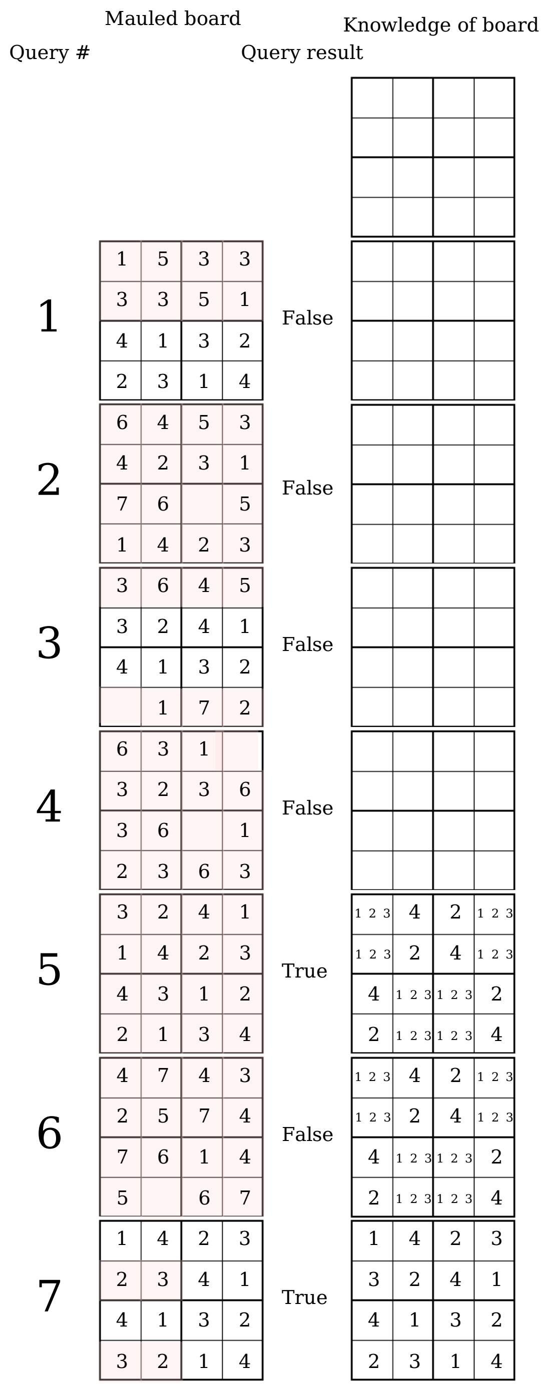


Figure 2.10: An example of an attack against the Sudoku format check. When the solver has narrowed a square to some candidate entries, multiple numbers are shown in a given square. Squares which are changed from their original values by a malleation are colored in light red.

a malleation which causes the resulting Sudoku board to remain valid, it can derive many additional constraints on the values of its model for the real plaintext. Other attacks took up to 33 queries, but have a similar pattern (deriving notable information from $True$ oracle results).

2.12 Discussion

We conclude this chapter with a discussion of extant related works as well as open problems uncovered as a result of this exploration.

2.12.1 Related Work

CCA-2 and format oracle attacks. The literature contains an abundance of works on chosen ciphertext and format oracle attacks. Many works consider the problem of constructing and analyzing authenticated encryption modes [BN00; Rog02; RS06], or analyzing deployed protocols, *e.g.,* [BKN04]. Among many practical format oracle attacks [Mau+15; Bar+12; Pod+18; RD10; YPM05; PY04; JS11; AF18; Kup+15; Gar+16a], the Lucky13 attacks [AP13; AP15] are notable since they use a *noisy* timing-based side channel.

Automated discovery of cryptographic attacks. Automated attack discovery on systems has been considered in the past. One line of work [CSP16], [Pha+17] focuses on generating public input values that lead to maximum leakage of secret input in Java programs where leakage is defined in terms of channel capacity and Shannon entropy. Unlike our work, Pasareanu et al. [CSP16] do not consider an adversary that makes *adaptive* queries based on results of previous oracle replies. Both [CSP16] and [Pha+17] assume leakage results from timing and memory usage side channels.

Using solvers for cryptographic tasks/model counting. A wide variety of cryptographic use cases for theory solvers have been considered in the literature. Soos *et al.* [SNC09] developed CryptoMiniSAT to recover state from weak stream ciphers, an application also considered in [COQ09]. Solvers have also been used against hash functions [MZ06], and to obtain cipher key schedules following cold boot attacks [AKMY10]. There have been many model counting techniques proposed in the past based on universal hash functions [GSS06; Zha+16]. However, many other techniques have been proposed in the literature. Several works propose sophisticated multi-query approach with high accuracy [SM19; CMV16a], resulting in the `ApproxMC` tool we use in our experiments. Other works examine the complexity of parity constraints [Zha+16], and optimize the number of variables that must be constrained over to find a satisfying assignment [Ivr+15].

2.12.2 Open Problems

In this work we developed novel techniques for automating the discovery of chosen ciphertext attacks on symmetric encryption scheme. We presented experimental results from our software, demonstrating that it generates successful attacks against both real-world and contrived format oracles. We view these initial results primarily as a demonstration of *feasibility*: our work leaves significant room for further optimization, some of which we apply and validate in Chapter 3.

Our work elucidates a number of open problems. In particular, we proposed several possible further optimizations to our tools which warrant exploration. Additionally, while we demonstrated the viability of our model count optimization techniques through empirical analysis, these techniques require theoretical attention. Our ideas may also be extensible in many ways: for example, developing automated attacks on protocols with side-channel leakage; on public-key encryption; and on “leaky” searchable encryption schemes, *e.g.,* [Gru+18]. Most critically, a key contribution of this work is that it poses new challenges for the solver research community, which may result in improvements both to general solver efficiency, as well as to the performance of these attack tools.

Chapter 3

McFIL

This chapter is based on work originally published in Usenix Security 2023 [ZCG23].

For hundreds of years, the history of what could be called cryptography was centrally focused on protecting the confidentiality of a message in transit or stored in some way, whether written on paper, spoken, or typed into an Enigma machine, for example [Sma16]. However, since the 1980s, there has been a monumental expansion of how confidentiality could be achieved and the capabilities of the systems designed to protect it. Three particular efforts, originating in the research literature, are the principal drivers of this expansion. The first is secure multi-party computation (MPC), generally credited to Yao in the early to mid 1980s [Yao82; Yao86a]; Zero-Knowledge Proofs (ZKP), originated by Goldwasser, Micali, and Rackoff in 1985 [GMR85]; and Fully-Homomorphic Encryption (FHE), initiated by Gentry *et al.* in 2009 [Gen09]. Each of these ideas have become full-fledged areas of research with multitudes of advancements since their beginnings, and they share one particular thing in common: they all provide confidentiality of data used in their protocols *plus* some additional desirable traits such as collaborative private program execution (MPC), homomorphism over addition and multiplication allowing arbitrarily complex arithmetic programs to be evaluated on ciphertexts (FHE), or the capability to guarantee, or at least imply with overwhelming probability, certain predicates of the plaintext given only the ciphertext (ZKP).

Given that the capabilities and usage of cryptographic confidentiality are so rapidly evolving, in order for the goals of usable cryptography for data confidentiality to be achieved, we must pursue these emerging areas in addition to more classical ones such as the chosen-ciphertext attacks of the prior chapter. This chapter covers these three distinct classes of private computation (MPC, FHE, and ZKP) and our approach to providing automated analysis based on our definition of privacy loss, which is built around the information leakage *inherent to* the functionality in question.

Functionality-inherent leakage (FIL) is a universal characteristic of systems which compute over private data. Modern cryptography, that is, secure multiparty computation, fully-homomorphic encryption (FHE), and zero-knowledge proofs [Ish+07], are all in essence computations performed on encrypted data, whether directly (FHE), collaboratively over a network (MPC), or in a prover-verifier model (ZKP, i.e. one party "executing" by generating a proof of some predicate(s), and another verifying them). The use of these cryptographic tools is growing steadily across the public and private domains, increasing the stakes of these systems' security and increasingly bifurcating the set of people who develop these systems from the set who use and rely on them.

FIL occurs when an adversary is able to observe or participate in computation over private data, and observe the outputs of this computation (or factors correlated with them). Naturally, given a computable function and even partial knowledge of some inputs and outputs, an adversary can infer *some*thing about the unknown inputs which induced observed outputs. The amount of information the adversary learns is determined by the structure and characteristics of the functionality in question.

In this chapter, we provide McFIL, an algorithmic approach for evaluating FIL within arbitrary functionalities, and an accompanying software implementation which can be used to determine the extent of leakage a functionality admits. Our approach does not exploit any cryptographic insecurity; rather, we leverage the unavoidable leakage inherent to underlying functions. The fact that cryptographic protocols can reveal information via correctly-evaluated outputs should be no surprise to cryptographers. However, these systems are now being more widely deployed by non-expert implementers who have few tools to analyze the functionalities they seek to evaluate. McFIL can derive inputs which *optimize* leakage, leveraging FIL to uncover confidential data, or provide probabilistic evidence of the absence of such an attack. It is our aim to provide a systematic way for prospective, non-expert users of cryptographic systems to evaluate functionalities they wish to compute securely, so they can make informed decisions on the risks of doing so even within cryptographically-secure schemes.

Our approach. We rely on a family of techniques called Model Counting, and we refer to our tool as McFIL for "Model Counting Functionality-Inherent Leakage." Most critically, McFIL is designed to quantify and optimize leakage *given only a description of the circuit to be implemented*, and does not require the implementer to assist the tool in understanding the functionality. This allows the tool to automatically derive a number of "attacks," including those not easily predicted by practitioners. For example, the classic Yao's Millionaires problem [Yao82] admits one bit of leakage per execution due to the functionality (greater-than comparison). Given only a description of the functionality, McFIL automatically derives inputs to uncover the other player's salary in approximately $log(n)$ sequential executions (Figure 3.1). Dual Execution MPC [MF06; HKE12] admits adversary-chosen one-bit leakage in an equality check protocol. McFIL derives a sequence of predicates of configurable complexity to uncover the honest party's input. We evaluate McFIL against an array of other functionalities as proofs of concept in §3.4, either completely recovering the honest parties input(s) or providing an equivalence class of candidate solutions many orders of magnitude smaller than the initial search space.

Leakage Explained. When considering novel functionalities for use in secure protocols such as MPC, FHE, or ZK, one must consider how the privacy of secret inputs will be maintained. Therefore, practitioners must consider the security of their schema but also what can be inferred from the outputs of the functionalities. Even with provably secure protocols, the confidentiality loss inherent to a given functionality may partially or even completely leak secret values without ever violating the security guarantees of a protocol. Proofs of security generally consider this form of leakage to be intrinsic, and arguments are made modulo whatever leakage may exist.

While functionality-based leakage may be unavoidable, it is quantifiable. Prior literature [CMS09; Mar+11; Mar+13a; Mar+13b] has applied various information flow and optimization techniques to this problem. However, quantifying information flow [CHM04] over programs is NP-hard and grows in the complexity of the program state space [KMM13]. As a result, these approaches have remained computationally infeasible in practice. In addition to these general results, bespoke analyses for individual protocols or leakage paradigms exist in the literature [Boy+12; HSV20; Alm+21], but these require manual analysis effort by experts which does not scale.

Contributions. We provide a practical methodology for *automatically quantifying* and even *optimizing over* inherent leakage of a circuit-based functionality. Our tool McFIL can be used to analyze privacy in MPC, FHE, or ZK. We generalize and extend a series of techniques developed in Delphinium (§2) in automating chosen-ciphertext attacks (CCA) in order to bring their work to a far broader class of functionalities.

In summary, our contributions are as follows. We describe McFIL, an algorithmic approach to quantifying and exploiting information leakage for a given functionality

(§3.3). We generalize and extend the Delphinium cryptanalysis framework [BZG20a] presented in Chapter 2 beyond its original domain of strictly-defined predicate functions, addressing key limitations of that work and resolving an open question from it (§3.3.1). We provide a software implementation [Zin23], as an artifact accompanying this submission and open-source tool. Our implementation encodes nontrivial domain knowledge to directly manipulate SAT instances and enable extensive parallelism (§3.3.2). Finally, we provide SAT instances generated from McFIL and our sample programs. To promote interdisciplinary research and give back to the SAT research community, we provide a wide array of problem instances generated by our tool as an additional artifact. The SAT community gathers such instances as benchmarks [HS00] to guide future SAT solver development, which will in turn expand the range and complexity of functions our tool can analyze within a given amount of computation time (§3.7.1).

3.1 Intuition: Maximizing Leakage

In the following two illustrative examples, we analyze simple functionalities to manually derive the results which McFIL automates. In the first example, we consider essentially the simplest possible functionality in an explanatory capacity.

In the second example, we also delve into a novel threat model we term the *multi-run adaptive* model. In this model, we assume the adversary or their delegates may execute a protocol multiple times. As protocols are often proven secure under the assumption of a single iteration or many non-colluding iterations, this assumption constitutes a model gap between the threats intended to be prevented and those we leverage against those protocols. We acknowledge that this gap is non-trivial, but with it, we demonstrate attacks against otherwise private functionalities. Without

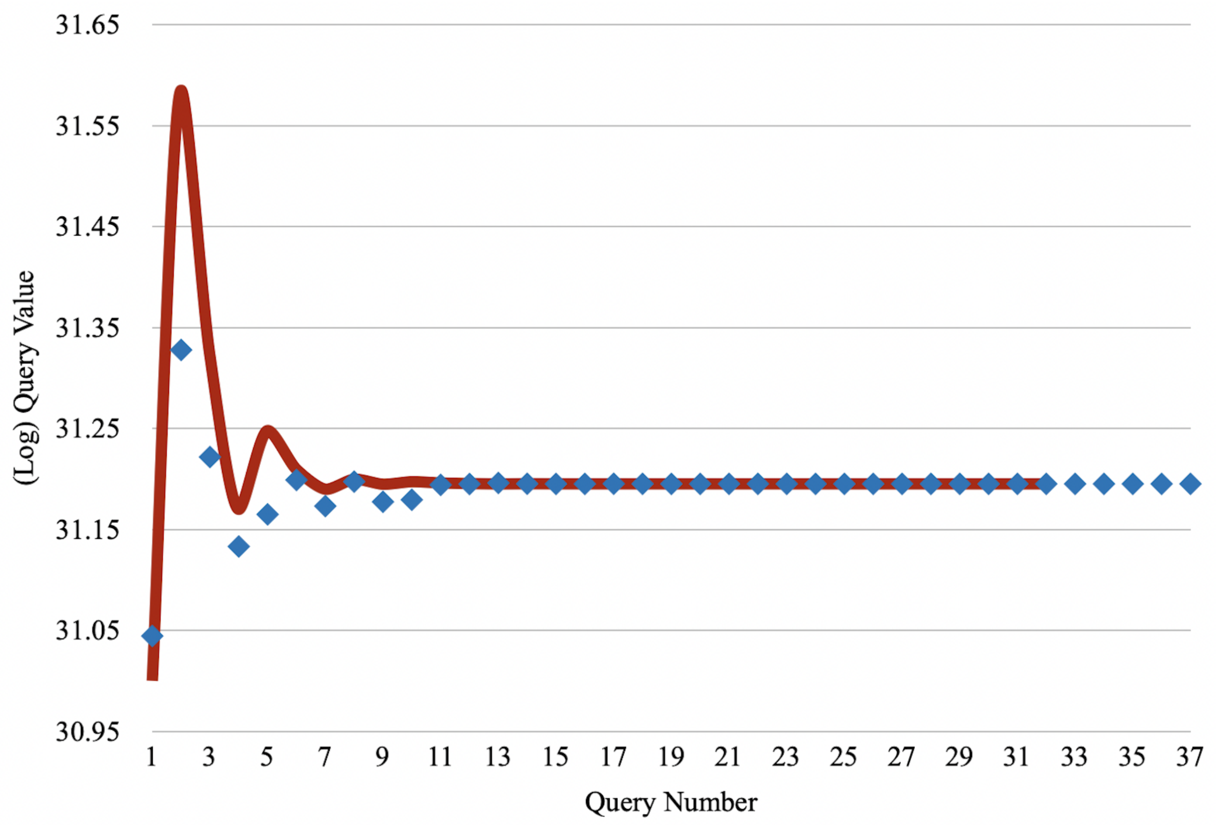


Figure 3.1: In a fully-automated analysis of Yao's Millionaires, McFIL "discovers" binary search. Blue diamonds are adversary inputs, true binary search shown in solid red.

this assumption, McFIL remains useful to analyze and quantify confidentiality loss as a result of functionality-inherent leakage, and to recover private inputs wherever single-iteration attacks are admitted. With it, however, McFIL is able to recover partial or complete secret inputs by leveraging functionality-inherent leakage alone.

3.1.1 A Minimal Example: Binary AND

Consider a simplistic two-party MPC (2PC) protocol which securely computes the functionality $\mathcal{F}$ consisting only of a Boolean *AND* gate ($\wedge$).

$$\mathbb{Z}_2 = \{0,1\}$$

$$\mathcal{F} : \mathbb{Z}_2 \times \mathbb{Z}_2 \to \mathbb{Z}_2 = \wedge$$

$a \in \mathbb{Z}_2$: honest party's input

$b \in \mathbb{Z}_2$: adversary's input

$x \in \mathbb{Z}_2$: output ($x = a \wedge b$)

Maximizing Leakage. After normal execution, the adversary should have no knowledge of a. However, knowing $\mathcal{F} = \wedge$, the adversary may *a priori* choose their input $b = 1$ to gain complete knowledge of a upon learning x:

$x = 0 \implies a \wedge 1 = 0 \implies a = 0$

$x = 1 \implies a \wedge 1 = 1 \implies a = 1$

The choice of $b = 1$ *maximizes the information* gained from x. Had they chosen $b = 0$, $x = 1$ would have become impossible for the AND functionality, and thus the output would provide no information about a. That is, x would be entirely independent of a, and a would have remained indistinguishable from a random bit to the adversary. By choosing b optimally, the adversary learns a despite the security of the MPC.

3.1.2 A Further Example: Yao's Millionaires

Now, consider the classic 2PC example of the Millionaires problem [Yao82; Yao86a]. Two parties (A, B) wish to compare their wealth (say, as 32-bit integers a and b) without disclosing anything aside from the predicate result of $a < b$.

$\mathbb{Z}_{32} = \{0, \ldots, 2^{32} - 1\}$

$\mathcal{F} : \mathbb{Z}_{32} \times \mathbb{Z}_{32} \to \mathbb{Z}_2 \;=\; <$

$a \in \mathbb{Z}_{32}$: honest party's input

$b \in \mathbb{Z}_{32}$: adversary's input

$x \in \mathbb{Z}_2$: output ($x = a < b$)

Maximizing Leakage. In this example, no single adversary input will uniquely constrain the honest party's input (at least, not in expectation over a uniform distribution of possible a).

Now, consider $b = 2^{31} - 1$. If $x = 0$ (`false`), the most significant bit of the honest input must be 1 as $a \geq 2^{31}$. Equivalently, if $x = 1$, then $MSB(a) = 0$. Assuming uniform a, in expectation the adversarial input $b = 2^{31} - 1$ eliminates half of the candidate solutions for a, providing the adversary a single bit of information.

Extending Leakage. On its own, this single bit of information out of 32 bits is potentially unimportant. The number of possible solutions for a remains large, and the adversary has no way to distinguish between them.

Intuitively, we next consider: *what if the adversary can try again?* Although this extends beyond the original threat model of many secure protocols, it does so with pragmatism: one can imagine settings where a protocol may be executed multiple times e.g. in a client-server model, when attempt-limiting protections are missing or evaded, or when an honest party may unwittingly interact with multiple colluding adversaries.

Maximizing Multi-Run Adaptive Leakage. Given additional attempts (or "queries," following [BZG20a]), a binary search emerges from the Millionaires comparison functionality. When the protocol may be repeated, a can be completely uncovered in $log(|\mathbb{Z}_{32}|) = 32$ queries, violating confidentiality despite a secure protocol, and making *optimally-efficient use* of the newly-assumed threat model of multiple queries.

This should be unsurprising to cryptographers: if security is proven modulo leakage, and leakage is allowed to grow, naturally security may be compromised. However, automatically analyzing and quantifying leakage over many queries is useful in two ways. First, the more intuitive: if a protocol does admit some method to retry, for example a smart contract [Woo+14] which may be repeatedly executed or an online oracle which does not sufficiently authenticate users, a multi-run leakage amplification attack may be possible. In this case, McFIL can be used to derive a set of queries to efficiently exploit leakage. Second, the multi-run setting can be useful in analyzing protocols which do not admit multiple attempts in that McFIL can compute an average leakage statistic per query. For arbitrary protocols, the true extent of leakage may be unknown; McFIL provides a way for practitioners to evaluate functionalities before choosing to encode them into secure protocols. As demonstrated in Figure 3.1, McFIL approximately rediscovers this binary search attack to uniquely identify the `target` secret input.

3.1.3 Overview of McFIL

Here we provide intuition as to how McFIL uses SAT solving to achieve automated leakage maximization. Refer to Figure 3.2 for a visualization of our automated workflow.

Building blocks: Delphinium. We previously applied SAT solving to the problem of Chosen Ciphertext attacks (CCAs) in Delphinium (Chapter 2) [BZG20a]. The methodology formulates CCAs as a sequence of optimization problems over the outputs of a format oracle, automating attacks against classic and novel oracles to exploit CCA vulnerabilities.

Format oracles are predicate functions which determine if an input is well-formed or not, and Delphinium exploits this leakage to decrypt. In this chapter, we formulate generalized functional leakage as a similar sequence of optimization problems. The main contributions, then, are the necessary adaptations to generalize the prior approach from predicates (1-bit outputs) to arbitrary n-bit output functions while maintaining concrete efficiency in real-world target functionalities. Further, Delphinium requires its format functions to be well-defined (e.g. for all possible inputs, either $true$ or $false$

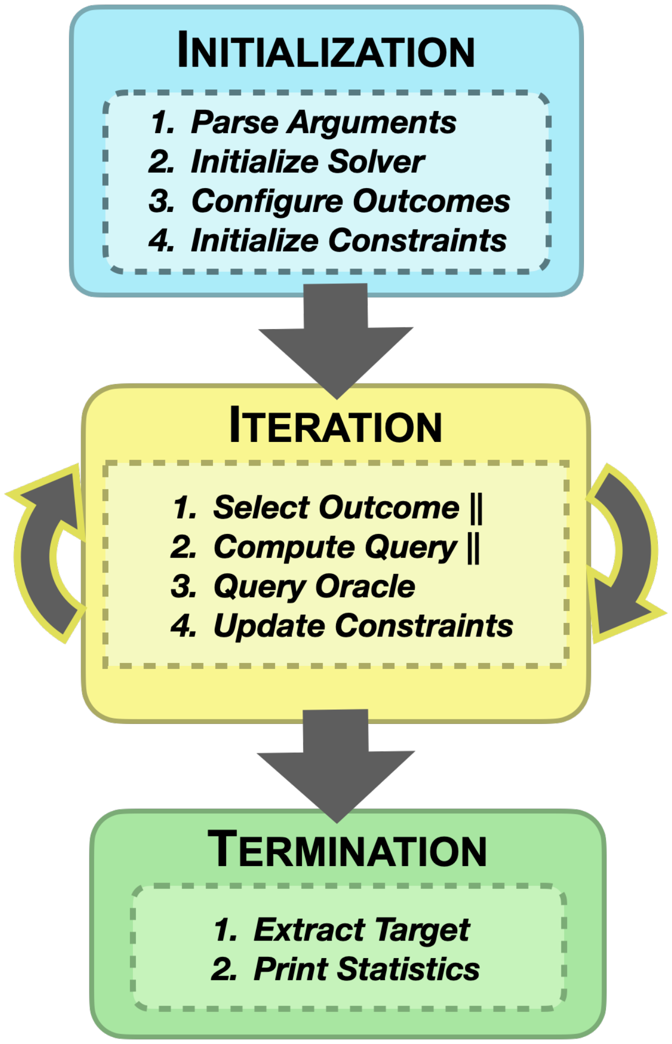


Figure 3.2: McFIL workflow; || denotes parallelization

is returned). To accurately capture the breadth of arbitrary functions, we release this requirement using a novel, concretely-efficient heuristic described in §3.3.1.

Initialization. To initialize McFIL, a Boolean circuit representation of a function functionality must be provided. McFIL uses this formula to determine the range of possible outputs of the functionality and to configure SAT instances. Compilers exist to facilitate this process [MB08], and McFIL offers a Python3 Domain-Specific Language (DSL) to facilitate encoding (refer to §3.6). These instances contain symbolic bits representing the "target" secret input(s) to be uncovered (`target`), and symbolic bits representing the adversary's "chosen" input(s) (generated by McFIL) or "queries" (`chosen`). This gives the solver freedom to choose `chosen` inputs subject to the optimization constraints we apply.

Iterative Solution Elimination. In order to uncover secret input(s) in an automated, multi-round analysis, McFIL requires access to a concrete instantiation of the functionality. Whether implemented as a test shim for local execution (as we provide in our implementation) or e.g. a live instance of a secure protocol, this instantiation must compute the functionality given the adversary's query and the true secret input(s) of the honest parties, and provide the result back to McFIL. We refer to this as the *function oracle*, and each iteration corresponds to e.g. a single MPC evaluation in the *multi-run adaptive* threat model.

Using a SAT solver, these two components alone are sufficient to iteratively constrain the search space of the secret input(s). However, the queries generated at each iteration would be arbitrarily drawn from the set of satisfying models within the solver – degenerating to a brute-force guessing attack within an exponential search space. What remains is to optimize the *profitability* (expected number of eliminated models [BZG20a]) of each query.

Optimizing Queries. A truly optimal leakage maximization would consider all possible numbers and contents of queries. However, due to the underlying complexity of the problem, this approach fails in practice. Existing leakage analysis work in side channels [Pha+17] attempts to analyze a sliding window of multiple query-like values, and reaches computational limits in 8-bit secret domains. Therefore, we consider only maximizing at each iteration: a greedy-optimal approach.

In order to maximize profitability at each iteration, we employ Max#SAT to *simultaneously maximize* the number of satisfying solutions for `target` corresponding to all possible outputs of the function under test. McFIL then iteratively derives function inputs which imply partitions of the solution space, and eliminates one of the subsets of each partition based on the result from the function oracle.

Max#SAT, described in detail in §3.2, is a counting-maximization analogue to Boolean Satisfiability (SAT) which can be efficiently, probabilistically approximated as demonstrated by Fremont *et al.* [FRS17]. Crucially, these approximation algorithms can themselves be efficiently encoded into SAT constraints [BZG20a]. By using these approximations as constraints in SAT instances, McFIL probabilistically and approximately ensures that a large number of satisfying solutions exist for given symbolic variables.

Extracting a Model. With all constraints in place and an unknown `target`, a satisfying solution (or "model") for `chosen` is extracted from the solver. The maximization constraints ensure that `chosen` is selected approximately optimally: for each possible output of $\mathcal{F}$(`chosen`,`target`), many candidate models for `target` exist.

Given only a symbolic representation of `target`, $\mathcal{F}$ cannot be used to compute a concrete result. However, when the concrete protocol instantiation – containing knowledge of the true value of `target` – is executed with the adversary's `chosen` input, a single outcome `result` is returned. Then, all candidate solutions for which $\mathcal{F}$(`chosen`,`target`) $\neq$ `result` may be eliminated. Critically, due to the maximization constraints, this set of eliminated candidates will be large.

Eliminating Candidates. After each query, all candidate models for `target` *inconsistent* with each (`chosen`,`result`) pair are eliminated. That is, they are no longer satisfying solutions for `target` in the evolved constraint system. This leads to a greedy algorithm in expectation, iteratively reducing the `target` search space in maximized increments.

Beyond Predicates. Problematically, the Delphinium algorithm is restricted to predicate functions, and intrinsically requires that for all `chosen`, every possible output for $\mathcal{F}$ is reachable. We refer to this informally as *completeness*:

$$\forall \texttt{chosen}, \texttt{result}\ \exists \texttt{target}$$

$$s.t.\ \mathcal{F}(\texttt{chosen}, \texttt{target}) = \texttt{result}$$

This completeness restriction is often trivial for predicates. While this is sufficient for chosen-ciphertext attack discovery [BZG20a], as the relevant padding/format functions are generally predicates which determine message validity, arbitrary functionalities are not necessarily so amenable. Further, Delphinium requires the *operator* to carefully define Boolean formulae to adhere to this notion of completeness, requiring additional effort and expert insight. We describe this challenge and our approach to generalize and extend Delphinium in §3.3.1.

Negative Results: Demonstrating Security. Depending on the functionality, it may be impossible to differentiate any classes of candidates; McFIL detects this and provides a message that the attack may proceed as “brute-force.” This negative result can also be taken as an indication that leakage is bounded for the given functionality. If little enough of the private input is derived before brute-force is required, this may be taken as a probabilistic argument for the security of the functionality against leakage-based attacks.

$$(x1 \vee -x2 \vee x3) \wedge (x1 \vee x2 \vee -x3) \wedge$$
$$(-x1 \vee -x2 \vee -x3) \wedge (-x1 \vee x2 \vee x3)$$

Figure 3.3: CNF of $x1 \oplus x2 = x3$

3.2 Technical Background

In this section, we discuss SAT, its extensions, and their relative complexity. We also highlight software tools solving or approximating these problems which have emerged from the SAT research community, and describe their relevance to McFIL. Finally, we review the limited past works which have broached function-inherent leakage and faced computational-feasibility limitations.

3.2.1 SAT and SMT

Boolean Satisfiability. SAT is a widely-known NP-complete problem. SAT takes as input a Boolean formula which relates a set of binary input values through Boolean operations. A solution to SAT is an assignment (also called a model, solution, witness, or mapping) of `true` and `false` (eq. 1 and 0) values to the Boolean inputs which causes the formula to evaluate to `true`. SAT formulae are commonly organized to include only And, Or, and Not operations in Conjunctive Normal Form (CNF) [Tse68]. Any propositional formula can be efficiently converted into CNF preserving satisfiability, although conversion may introduce a linear increase in formula size [Tse68]. In CNF, *literals* represent binary input values, and may be negated (denoted with −). These literals (negated or otherwise) are grouped into disjunctions (Or), which are themselves grouped into a single conjunction (And). In short, CNF is an "and of ors." Refer to Figure 3.3 for an example.

SAT Solvers. SAT solvers are tools designed to determine the (un)satisfiability of Boolean formulae. SAT solvers often require CNF, and provide a *model* in the event the formula is satisfiable. The model is not guaranteed to be unique (and often is not).

Due to the generality of NP-complete problems [Coo71], SAT solvers are powerful tools. Since 1962, the DPLL [DLL62b] method of backtracking search has served as the core tool in SAT solving, with the more recent development of Conflict-Driven Clause Learning (CDCL) [SS96] in 1996 aiding in optimizing the search for a satisfying model or contradiction indicating UNSAT.

Satisfiability Modulo Theories. Given a SAT solver, the task remains to translate problems of interest into Boolean formulae. In order to aid translation, SMT solvers were developed. SMT solvers add expressive domains of constraint programming to SAT such as arithmetic and bitvector (bitwise operations beyond single Boolean values) logic. SMT solvers such as Z3 [MB08] have enabled solving problems ranging from program verification [CDE+08; Sho+16; GLM12] to type inference and modeling, and more.

3.2.2 Extensions to SAT

Despite the expressive power of SMT solvers, program analysis alone is insufficient for McFIL as we derive optimizations over the target circuit. As we *maximize* over a solution space of program results, McFIL generates instances of Max#SAT (described in this section) from the Boolean formula describing the functionality under test. Intuitively, Max#SAT is a strictly more complex problem than SAT, but can be approximated using the recent technique of Fremont *et al.* [FRS17] and a SAT solver. In this section we provide technical background describing SAT through Max#SAT.

SAT to #SAT. #SAT, pronounced “sharp SAT,” is the counting analogue to SAT. Also

Definition 4: $Max\#SAT(\phi, \overline{X}, \overline{Y}, \overline{Z}) = \overline{X}_{max}, max$

Input: ϕ: Boolean formula;
$\overline{X}, \overline{Y}, \overline{Z}$: Vectors of Boolean inputs to ϕ

Output: $\overline{X}_{max}$: Assignment (concrete Boolean values) of variables in $\overline{X}$ for which the number of satisfying solutions to variables in $\overline{Y}$ is maximized and at least one satisfying solution exists in $\overline{Z}$;
max: Number of satisfying solutions for $\overline{Y}$
$(0 \leq max \leq 2^{|\overline{Y}|})$

referred to as *model counting*, #SAT asks not only *if* a satisfying model exists, but *how many* such models exist. The result lies in a range from 0 for an unsatisfiable formula, to 2^n for a (completely unconstrained) formula over n bits. #SAT is #P-complete, at least as difficult as the corresponding NP problem, but this phrasing belies its complexity: Toda demonstrated PH $\subseteq$ $P^{\#P}$ [Tod91], that a polynomial-time algorithm able to make a single query to a #P oracle can solve any problem in PH, the entire polynomial hierarchy (which contains both NP and co-NP) [Sto76].

#SAT to Max#SAT. Max#SAT (denoted in Definition 4) is the optimization analogue to the #SAT counting problem. As defined by Fremont *et al.*, Max#SAT takes a Boolean formula, denoted ϕ, over Boolean variables divided into three notional subsets $\overline{X}, \overline{Y}, \overline{Z}$. A solution to Max#SAT provides a model for the variables in $\overline{X}$ such that the count (number of models) of the variables in $\overline{Y}$ is maximized and at least one model exists for the variables in $\overline{Z}$. ($\overline{Z}$ exists to allow variables to remain in the SAT instance, yet outside the optimization constraints). A Max#SAT solution is particularly useful when $\overline{X}$ refers to input variables to the formula, meaning that a model for $\overline{X}$ could be provided as an input to a program represented by the circuit ϕ. Correspondingly, $\overline{Y}$ should be configured to contain variables of interest for maximization such as those representing quantitative information leakage, probabilistic inferences, or program-synthesis values [FRS17].

Approximating Max#SAT. Max#SAT is at least as complex as #SAT [FRS17; Tod91], but as Fremont *et al.* demonstrates, this does not prevent approximating Max#SAT using a number of calls to an NP oracle – or in concrete terms, a SAT solver. To approximate Max#SAT, Fremont *et al.* rely on a sampling technique first described by Valiant [Val79b], and expanded upon (and implemented in software) by Chakraborty and Meel *et al.* [CMV16b] and Soos [SM19]. By applying *almost-uniform* hash functions [Val79a; SM19], which representatively sample a domain with error, a non-uniform search space can be proportionately sampled with bounded error to enable a more feasible count. Crucially, these hashes can be efficiently sampled and applied within a SAT solver [Cha+14b]. This approach is central to McFIL: using *almost-uniform* hash functions as constraints within the formula, large classes of models for `target` can be identified and iteratively eliminated.

3.2.3 Prior Work

Functionality-based leakage is a known problem, although practical quantification methods are lacking in the literature. Clarkson *et al.* [CMS09] analyze adversary knowledge of private MPC inputs using quantified information flow, but provide only a theoretical treatment of the problem. This idea was later explored by Mardziel *et al.* [Mar+11; Mar+13b; Mar+13a] in their attempt to maximize adversary knowledge through probabilistic polyhedral optimization. Although their solution is theoretically robust, Mardziel *et al.* note the "prohibitive" computational cost of their approach in practice [Mar+13b].

SAT solving has also been applied to MPC in the recent literature, however, with particular focus on the *intermediate values* generated during protocol execution. These intermediate values may leak some degree of information, which has been quantified using information flow [Ras+13] and language-based formalization methods [Alm+18].

Further, when intermediate values can be determined to leak only *negligible* information, prior work has shown they may be computed "in-the-clear" as an optimization [Ras+13].

Quantitative information flow, program analysis, and other automated approaches have also been applied in the detection and mitigation of side-channel leakage vulnerabilities. These vulnerabilities are similar to function-inherent leakage in that they provide additional signal which may compromise private inputs based on the (otherwise secure) execution of a function. A recent systematization [Buh+22] of side-channel detection literature enumerates various techniques ranging from micro-architectural modeling or even reverse engineering to program-analysis-like tools to analyze hardware descriptor languages for potential side channels. One such work, SLEAK [WHK14], computes a statistical distance between secret values and active intermediate values in the processes using a full-system simulator. The information-theoretic metric of leakage they compute is likely correlated with the leakage which McFIL is able to uncover and maximize, however, their approach requires hardware simulation and a compiled binary, neither of which may be available or even relevant to target functionalities intended to be executed e.g. within an MPC.

3.3 McFIL

In this section, we describe our primary contribution, McFIL. We introduce the novel `SelectOutcomes` prioritization subroutine which extends the Max#SAT-based approach beyond *complete* predicate functions and enables its use in the domain secure protocols such as MPC, FHE, and ZK.

Attack Model. An ideal approach to identifying leakage would allow for the quantification of useful leakage after one round of execution. For novel function circuits,

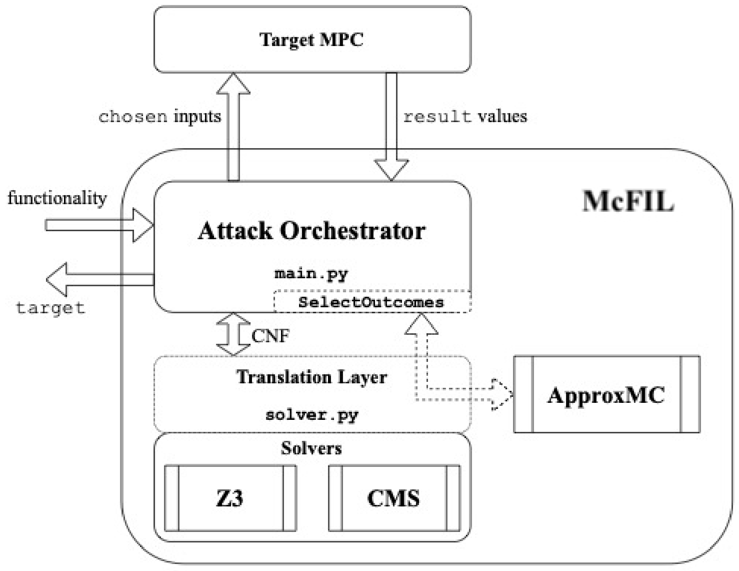


Figure 3.4: Architecture of McFIL

this analysis can inform operational security requirements and privacy considerations. However, in many cases a functionality will be executed multiple times on identical (or related) inputs (this can occur in some protocols by design; alternatively it may occur through deception, corruption of honest parties, or a failure of access control). Ensuring the safety of all private inputs in these cases can also be seen as a form of defense-in-depth. For this setting, a *multi-run adaptive* model allows McFIL to extract more information about the privacy implications of a given Boolean circuit, whether the goal is to evaluate the safety of allowing untrusted parties to execute the protocol repeatedly (to automate an attack), or simply to to quantify and bound overall leakage.

System Architecture. Figure 3.4 depicts the architecture of McFIL. Our tool integrates a custom CNF manipulation toolkit (`solver.py`) which efficiently represents constraints and orchestrates parallel solving instances.

3.3.1 Beyond Predicate Functions

In Delphinium [BZG20a], a format oracle classifies inputs as either valid or invalid according to a format specification and a predetermined fixed input length l. This requires format specifications to be *complete* and *deterministic*. Completeness requires that for every possible bit string of length l, the format function returns exactly one of `true` or `false`. Determinism requires that any input to the format function will always be classified the same way. As a result, the format function defines a two-set partition of the space of bit strings of length l, with the two sets corresponding to `true` and `false` under the format, respectively. Taken together, the requirements of Delphinium ensure that at each iteration, the SAT formula will remain satisfiable as long as a query exists which can differentiate at least some `true` and `false` partition elements.

Challenge Intuition. Clearly, arbitrary functions may be more complex than predicates. To address this, a natural first idea is to implement a third possible response (in addition to `true` and `false`), e.g. `error`. By introducing an "error" class, outputs could be simplified and the completeness requirement removed. However, adding a third output class raises a unique challenge previously left to future work.

McFIL seeks to *simultaneously maximize* the number of secret inputs which correspond with *all* n output classes, such that each round, up to $\approx \frac{1}{n}$ of candidates may be eliminated in expectation. However, with the introduction of an error class and relaxed function definitions, the guarantee of simultaneous satisfiability is lost. For example, it may be that an input exists where some candidates would evaluate to a given output, and the rest to `error`, but none to another output. Delphinium requires simultaneous satisfiability of all classes, and so the resulting `UNSAT` result immediately halts progress.

This idea of three output classes is also only a specific instance of a much more general challenge. For arbitrary functions with n-bit outputs, predicates represent only $n = 1$. For functions with arbitrary output length $n > 1$, the number of simultaneously satisfiable classes is entirely dependent on function itself; as a result, naïve simultaneous maximization fails or reaches computational limits.

A First Step: `DeriveOutcomes`. Functions with n-bit outputs have a maximum of 2^n possible output values. However, for many functionalities the actual number of possible outputs is much lower. A natural example is a classification function: each class may be described by a long bit-string (a long output), but only a few classes may exist.

Detecting and exploiting these cases contributes to McFIL's practical applicability. At each iteration of an attack, the number of possible outputs defines the number of simultaneous maximization constraints which must be satisfied. Therefore, we provide the `DeriveOutcomes` subroutine, which uses the SAT solver to enumerate the possible outputs of the functionality. We perform this step one time at initialization (the *Configure Outcomes* step in Figure 3.2), dramatically reducing constraint system size from the 2^n maximum in many cases.

Analysis of `DeriveOutcomes`. `DeriveOutcomes` is executed once at the beginning of the McFIL workflow. It iterates over all 2^n possible outputs of the target n-bit output functionality in order to eliminate unreachable outputs. Although this can be concretely expensive, many functions with n-bit outputs do not entirely cover their range; eliminating large classes of unreachable outputs dramatically improves performance for the remainder of McFIL's steps. Additionally, this step can be skipped, and partial completion still benefits computation time significantly.

Optimizing Leakage: Straw-man Solution. With the set of possible outputs derived, it remains to eliminate large classes of candidates across these output classes at each iteration. Otherwise, the attack is approximately brute-force and therefore highly inefficient. If all $N \leq 2^n$ classes cannot be simultaneously differentiated from one another by a single query, the straightforward next step would be to differentiate *as many classes as possible*.

Here we reach the next challenge: choosing an optimal subset of output classes. The number of subsets of a set of N values is 2^N, i.e. 2^{2^n} for n-bit outputs. Even if `DeriveOutcomes` eliminates many outputs to reduce N, the exponential size remains problematic for performance. This combinatorial space is far too large to efficiently iterate through – worse, doing so would be required at *each iteration* due to evolving constraints. As a result, in order to support non-predicate functions which may contain mutually-exclusive outcome classes, optimality must be sacrificed in favor of an efficient heuristic which performs well in practice.

Solution Intuition. To avoid a combinatorial search, a heuristic must be employed. However, the accuracy of the heuristic directly impacts the profitability of the resulting query, and thus warrants specific analysis and consideration. The intuition for this heuristic (which we refer to as `SelectOutcomes`) is to remove outcomes from the simultaneous maximization constraint system while minimally affecting profitability.

To achieve this, we perform Model Counting (#SAT) on the solution space of each outcome individually. Each result corresponds to the maximum number of candidate models which *might be eliminated* by including that outcome in the simultaneous maximization. We sort all outcomes by their individual sizes, and remove them in ascending order until a simultaneously satisfiability subset is found.

Of course, this approach may miss an ideal configuration of outcomes. However, avoiding combinatorial search is necessary, and in our evaluation (§3.4), we demonstrate that profitability remains sufficient to discover efficient attacks.

Algorithm 5: `SelectOutcomes`

```
Input: Solver: SAT solver, Pool: multi-processing pool, $\mathcal{F}$: formula for
        functionality, $\mathbb{O}$: set of outcomes from DeriveOutcomes
Output: $\mathbb{O}^*$: set of mutually-compatible outcomes w/many candidate target,
        for use in maximization constraints
$\mathbb{O}^* \leftarrow \emptyset$;
for $i \leftarrow 1$ to $|\mathbb{O}|$ do
    $O_i \leftarrow i$th element of $\mathbb{O}$;
    Solver.constrain($O_i = \mathcal{F}$(chosen, target));
    // generate CNF for ApproxMC
    $\phi_i \leftarrow$ Solver.cnf();
    // remove constraint for next iteration
    Solver.pop();
    // call ApproxMC in parallel
    Pool.apply_async(ApproxMC, $\phi_i$);
// for each outcome and its ApproxMC count
for ($O_i$, $cnt$) in Pool.results() do
    if $cnt > 0$ then
        // satisfiable single outcome
        $\mathbb{O}^*$.add(($O_i$, $cnt$));
$\mathbb{O}^* \leftarrow$ sort $\mathbb{O}^*$ by $cnt$ ascending;
while $|\mathbb{O}^*| > 1$ and not Solver.satisfiable($\mathbb{O}^*$) do
    $\mathbb{O}^*$.drop_first();
// largest simultaneously satisfiable set
return $\mathbb{O}^*$;
```

`SelectOutcomes`. To realize this heuristic, documented in simplified form in Algorithm 5, we employ a powerful tool from the recent SAT solving literature: ApproxMC. ApproxMC [SM19] ("Approximate Model Counting") is a tool which takes a CNF Boolean formula and rapidly provides an approximation of the formula's model count. ApproxMC can count complex formulae in a fraction of the time it takes to compute maximization constraints to count formulae. However, it cannot completely supplant maximization: ApproxMC is an external tool which uses sampling to provide approximate counts; there is no known way to efficiently encode iterative sampling *within the solver* as a constraint, and it is unlikely a method exists due to the data-dependent nature of the

ApproxMC sample-and-iterate strategy.

By iterating through the remaining satisfiable outcomes at each iteration of the attack and invoking ApproxMC, we ensure that simultaneous maximization still occurs among as many large classes as can be efficiently identified. Further, we employ process-level parallelism to amortize this sequence of ApproxMC calls. ApproxMC configuration parameters can also trade off single-instance computation time for accuracy if needed. Once a set of mutually-compatible outcomes is found, the attack proceeds with simultaneous maximization using Max#SAT. The resulting query is extracted and executed in the to produce a result. By design, this result tends to eliminate a large class of remaining candidate solutions. In the event the result corresponds with a removed outcome class, relatively little knowledge is gained. However, as the removed outcome classes are the smallest, the likelihood that an arbitrary query induces a removed outcome is minimized over a distribution of possible `target` values.

Algorithm 5 documents the `SelectOutcomes` heuristic. In the first loop, each outcome is individually constrained to generate a set of formulae using our CNF manipulation interface and the underlying SAT solver, resetting the solver after each iteration to individually test each outcome. Parallel tasks are dispatched to perform `ApproxMC` counts of the bits corresponding to the `target` variable in each formula (bit correspondence requires formula manipulation, omitted for clarity). These results for each outcome are sorted, and until a simultaneously-satisfiable subset is found (more complex than a satisfiability check, but omitted for clarity), the outcomes with the smallest number of candidate `target` solutions are eliminated. Finally, the usable subset $\mathbb{O}^* \subseteq \mathbb{O}$ is returned.

Analysis of `SelectOutcomes`. Delphinium avoids the need for any such heuristics by strictly requiring well-defined and inflexible problem statements. As a result of these strict requirements, the authors of Delphinium are able to formally prove, probabilistically and approximately, a greedy-optimal algorithm. McFIL enables a far broader scope of functionalities to be analyzed and attacked. Further, reducing the restrictions on the formulae input to McFIL reduces the operator effort and expertise required. By allowing significantly more flexibility in terms of functionality choice and reducing the modeling work required, McFIL sacrifices formal optimality for generality, performance, and practicality.

3.3.2 Implementation

Algorithm 6: McFIL Algorithm Overview

```
Input: $\mathcal{F}$: Boolean circuit with target and chosen input(s), $\mathcal{O}$: Oracle access to
       functionality with hidden target input
Output: $\mathcal{I}^*$: set of Boolean circuit inputs (vectors of Booleans) for chosen which
        maximize leakage of target in $\mathcal{F}$
$\mathcal{I}^* \leftarrow \emptyset$;
$out \leftarrow$ DeriveOutcomes($\mathcal{F}$);
while # of solutions for target > 1 do
    $\overrightarrow{sel} \leftarrow$ SelectOutcomes($\mathcal{F}, out$);
    $query \leftarrow$ Maximize(chosen, $\mathcal{F}, \overrightarrow{sel}$);
    $result \leftarrow \mathcal{O}(query)$;
    $\mathcal{F}$.AddConstraint($\mathcal{F}(query$, target$) = result$);
    $\mathcal{I}^*.add(query)$;
return $\mathcal{I}^*$;
```

McFIL consists of under 2 KLoC of new Python3 which leverages process-level parallelism at every opportunity. Each of the multiple linear parameter-sweep searches from the Delphinium software implementation is replaced with parallel execution, and any other code from the prior work is rewritten improving efficiency, clarity, and correctness. Parallelizing McFIL is non-trivial, requiring a reorganization of data-dependencies (CNF generation prior to solving) in the software pipeline of prior work. We introduce a CNF translation layer to readily convert SAT instances and even individual CNF clauses and literals between the two SAT solvers we employ, CryptoMiniSat and Z3, to reap the relative benefits of each (performance and flexibility, respectively). Our implementation includes a test shim for simulated execution of protocols, and a convenient command-line interface encapsulating numerous configuration options. It is available as open source software on GitHub [Zin23]. Algorithm 6 provides a high-level pseudo-code overview of McFIL using the subroutines described in Section 3.3.1.

CryptoMiniSat. CryptoMiniSat [SNC09] (CMS) by Soos *et al.* is a SAT solver designed for cryptographic use-cases. Specifically, CMS includes optimizations for rewriting and processing Xor operations which otherwise incur exponential overhead in the number of operands of a CNF representation. Specifically, an n-term Xor expands to 2^{n-1} CNF clauses.

We evaluate McFIL in §3.4 and provide wall-clock computation time to illustrate the practicality of attack generation when using CMS. We confirm the observation from Delphiniumthat SAT instances containing Xor-dense maximization constraints execute up to an order of magnitude faster in CMS than Z3, and for larger (longer-running) problem instances our parallelized query search offers additional multiplicative factors of time savings.

Z3. Z3 [MB08] is an SMT solver which provides extensive and robust software support for Boolean formula manipulation and solving. It is a general-purpose solver which supports arithmetic, bit-vector, and other common theories for ease of use in translating general problems to SAT. We use Z3 for its bit-vector theory support and other tooling. However, Z3 lacks key optimizations which accelerate solving the the particular Xor-dense maximization formulae we require. As a result, after formula preparation in Z3, we generate and export a CNF and use CMS for solving, and then recover results back into Z3 for the next round of constraint manipulation.

3.4 Evaluation

To evaluate McFIL, we assess our implementation against a range of functionalities, from simple motivating examples such as the classic Yao's Millionaires problem [Yao82] to the recent practical instantiation of MPC developed by researchers at Boston University with the Boston Women's Workforce Council (BWWC) to measure wage equity in a manner which preserved the privacy of participants' salaries [Lap+16]. The success of McFIL in partially or completely deriving confidential inputs across these functionalities demonstrates its usefulness as a tool for practitioners and researchers alike in performing privacy analysis of secure protocols.

3.4.1 Selected Functionalities

The following selected target functionalities are used to evaluate McFIL. Implementations of these functionalities are provided in the open source release using a simple Python3 domain-specific language to describe Boolean formulae. Samples of this DSL can be found in §3.6. We provide additional illustrative samples used in our evaluation in our open source release [Zin23]. These implementations may be useful for further analysis, or to serve as templates for translating new functionalities.

Evaluated Functionalities. Table 3.1 lists the evaluated functionalities and the search space size of each corresponding hidden `target` value. Relatively smaller search spaces were chosen compared with what is practically achievable in reasonable wall-clock time. This allowed evaluation of computation bottlenecks in McFIL while keeping overall evaluation runtime feasible for many randomized repetitions. McFIL outputs estimated leakage per query, useful for evaluating functionalities after only a single iteration of the tool. CNF size growth (in # $clauses$) across input sizes is demonstrated in Table 3.2 to inform extrapolations.

Table 3.1: McFIL Evaluated Functionalities

Functionality	`target`	**Leakage**
Yao's Millionaires	2^{32}	$2^{30\pm1}$
Dual Execution	2^{12}	$2^{10}-2^{11}$
Danish Sugar Beets Auction	$2^{14}-2^{56}$	up to 2^{54}
Bucketed Mean	2^{32}	$2^{23\pm1}$
Wage – Circuit Division	2^{36}	$2^{34\pm1}$
Wage – Standard Division	2^{36}	$2^{34\pm1}$
Mean Average	2^{8}	$2^{0}-2^{1}$

Refer to §3.4.1. Smaller domain sizes (`target`) were chosen to allow many randomized trials. Leakage conservatively estimates eliminated candidates per query, calculated in a single iteration of McFIL.

3.4.1.1 Millionaires Problem

The Millionaires problem introduced by Yao [Yao82] describes two millionaires, Alice (A) and Bob (B), who wish to compare their wealth without revealing it. Each has a corresponding input, their net worth represented as 64-bit integers a and b, respectively. The functionality to be computed in this simple example is the comparison operation less-than. Thus, at the end of the two-party computation, A and B learn the result of the predicate $a < b$, but learn nothing of each other's input.

This functionality clearly enables a binary search, and the resulting exponential decreases in the search space per query confirm the attack algorithm's approximate optimality (§3.6.1). We implement this functionality within `millionaires.py`. The attack is not particularly subtle, and could certainly be developed and executed manually. However, the example is illustrative, demonstrating the SAT solver rediscovering the known optimal attack without interaction or guidance.

Table 3.2: CNF Size in Clauses over Target Bits

Functionality	8-bit	16-bit	32-bit	64-bit
Yao's Mill.	47	95	191	383
Dual Exec.	875	3539	14243	57155
Danish.[*]	2746	6500	8701	14020
BM	457	561	769	1185
WCD	**	**	2178	5351
WSD	60	749	7705	42417
MA	191	399	815	1647

Refer to overview and descriptions in §3.4.1.

[*]*Due to encoding, the Danish Sugar Beets functionality was measured at 12, 28, 42, and 60 bits.*

[**]*The BWWC WCD function reorganizes division into a multiplication circuit which overflows for small bit-widths.*

3.4.1.2 Dual Execution

Dual Execution [MF06] is an MPC technique which enables conversion of semi-honest secure protocols into malicious secure-with-abort protocols incurring only a single bit of additional leakage. The technique involves both parties in a 2PC garbling a Boolean circuit and sending each other the circuit and necessary data for its evaluation. Both circuits are evaluated, and then a secure comparison protocol informs both parties if the outputs matched. If they do not, the protocol is aborted. The additional bit of leakage comes from the abort, which informs an adversary that the garbled circuit it sent to the honest party did not match the output of the one generated by the honest party – this knowledge can be leveraged to leak a bit of the honest party's private input upon each execution.

To implement this part of the adversary's input string is considered to be an encoded program. This program represents the divergent functionality the adversary may choose. For simplicity, we limit the adversary to affine transformations, which can be realized through a matrix multiplication. The complexity of function is entirely up

Table 3.3: Evaluation Results

Functionality	`target`	**Size**	**Queries**	**Mean**	**S.D.**	**Time**	**Speedup**
Millionaires	●	2^{64}	69 – 97	84.7	8.5	≈ 3m	2×
Dual Execution	●	2^{12}	17 – 80	34.9	12.9	≈ 0.5m	1.5×
Sugar Beets	◐	2^{28}	13 – 13	13.0	0.0	≈ 5m	4 – 5×
Bucketed Mean	●	2^{32}	35 – 58	36.7	6.4	≈ 6m	10×
Wage	◐	2^{36}	18 - 29	22.4	5.8	≈ 5m	1.5 – 2×
Wage (cir. div.)	◐	2^{36}	19 – 26	22.3	5.0	≈ 0.5m	1.5 – 2×
Mean Average	●	2^{8}	2 – 3	2.5	0.5	≈ 2.5m	0.8 – 1.2×

`target` uncovered ...partially ◐ ...completely ●

Size denotes the size of the `target` domain. Average time (in minutes) given for full attack, not per-query. Minimum and maximum queries listed with Mean and Standard Deviation. Average queries reported over ≈ 100 randomized trials. 'Speedup' denotes approximate wall clock time savings through parallelization.

to the adversary, however, allowing configurability is preferable in the dual execution setting where the honest party must be expected to believe they are executing an honest circuit rather than expecting an abort due to a mismatched equality check. The adversary then provides both its own input and the "code" the honest party will execute. McFIL is able to uncover all bits of the private input in very few queries (§3.6.1), even with the limitation to affine functions.

3.4.1.3 Danish Sugar Beets Auction

The first widely-known practical use of MPC was reported in 2009, when the Danish sugar beets auction was deployed as an MPC. The purpose was to replace the work of a trusted-by-necessity auctioneer with secure computation. The functionality takes in a set of buyer and seller orders to determine the *Market Clearing Price*. An order consists of a set of prices and the number of units (e.g. tons of sugar beets) a buyer/seller is willing to buy/sell at each price. The Market Clearing Price is the equilibrium price at which optimal volumes of sugar beets are exchanged. The functionality is characterized by a matching of buyer and seller prices weighted by the units at each price.

In this protocol, there are a configurable number of buyers and sellers, and we leverage this to test McFIL in the multi-party MPC setting (rather than two-party computation). When modeling adversarial participants, McFIL expresses all adversary inputs as a single vector of Boolean variables over which it determines structure and exploits leakage based on the functionality (Market Clearing Price calculation). This encodes the well-understood behavior of colluding malicious participants in an MPC. We implement this within `sugarbeets.py`.

3.4.1.4 BWWC

Collaborating with the Boston Women's Workforce Council (BWWC), Lapets *et al.* at Boston University recently developed and deployed an MPC system to detect gender-based wage inequity [Lap+16]. This system allowed companies to privately share aggregate statistics about employee roles, salaries, and gender. The goal was to computationally identify and measure gender-based wage discrimination without requiring any participating companies to directly demonstrate fault. Their design, robustly proven secure under standard assumptions, is an MPC which computes a simple statistical test among each comparable role across companies. By design, their system protects the privacy of each salary data point.

For this target, we developed two aggregation functions inspired by the functionality described. The first was a bucketed mean computation (`mean_buckets.py`) which averaged the two inputs and then returned a result representing the number of fixed-size buckets apart the query is from the updated mean. This bucketing procedure implements a non-predicate functionality in which many outcomes are simultaneously satisfiable, exercising the attack algorithm in this regard. The second averages two contributed salary data points (one honest input, one adversary) into a large aggregated salary, and then informs the querying adversary if their contributed salary is below the updated mean. This functionality enables the attack algorithm to derive a binary search for the honest input as demonstrated in the results section (§3.6.1).

We developed two forms of this second function, one (`wage_circuit_div.py`) with the division reorganized into a multiplication in the mean computation, and the other (`wage.py`) with regular integer division inside the solver. Division steps dominate the complexity of this functionality, as demonstrated by the difference between BWWC functions 2 and 3 in Table 3.1. Finally, we implemented a plain mean average (`mean.py`) function (BWWC function 4 in the Table) to evaluate a function with maximal (2^n) possible outputs for an n-bit `target`. Despite the small size of the secret in this case, computation time increased due to the large output domain as demonstrated later in this section.

3.5 A Note on Zero-Knowledge Range Proofs

Note on ZK Range Proofs. A range proof [Bün+18] is a computational proof that a value lies within a given range. Range proofs can be performed in zero-knowledge protocols, meaning that the verifier learns nothing of the value being tested, only the result of the computation. Such zero-knowledge proofs are useful in anonymous payment systems which require aggregate payment validation without betraying information about individual payments to the broader payment network [Sas+14]. These zero-knowledge proofs can be considered a 2PC between a prover (holding a secret value) and a verifier (holding a range to be tested against the value) wherein the functionality executed is the aforementioned range validation check and the privacy of the prover's value must be maintained. The logic of a range proof effectively matches that of the Millionaires problem, and so this target is included simply to highlight McFIL's capacity to analyze

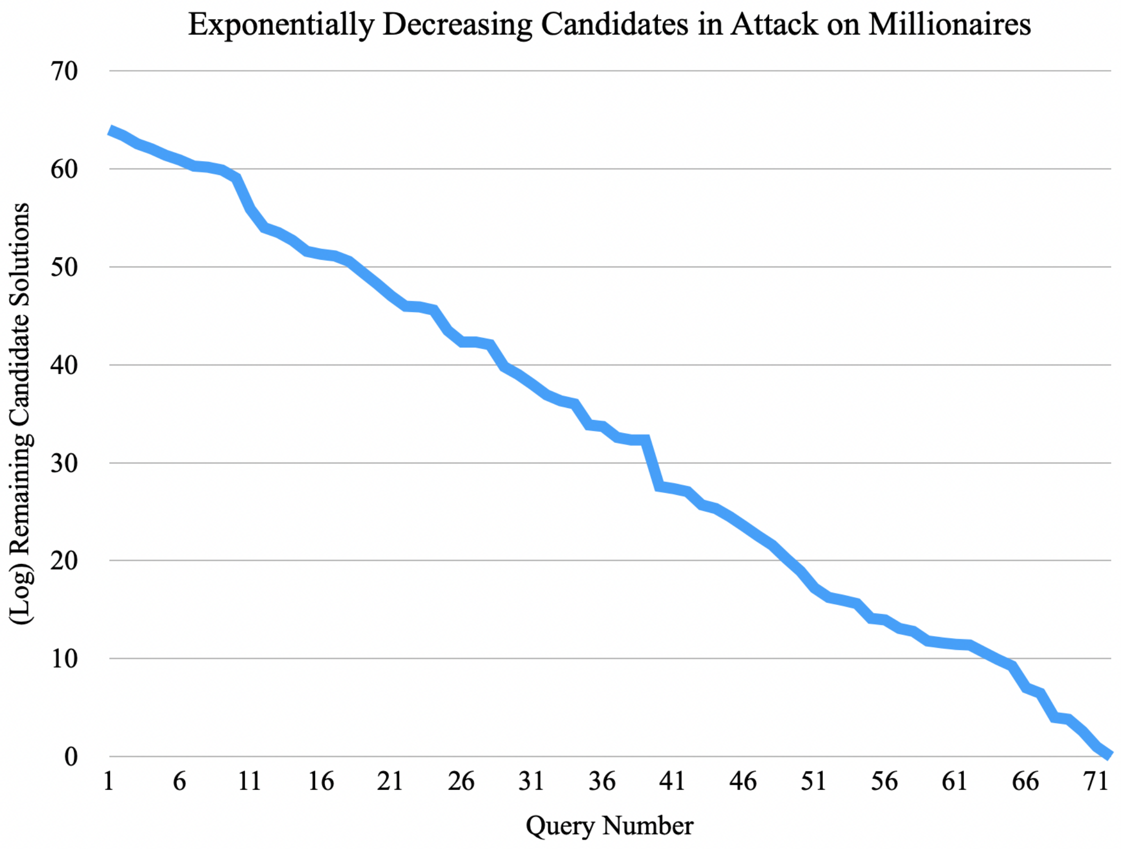


Figure 3.5: Remaining candidate solutions per query in a single trial of 64-bit Yao's Millionaires

a ZK protocol. We offer our software implementation as a tool to researchers and developers of novel ZK protocols as a method to evaluate privacy loss.

3.6 Solver Domain-Specific Language Examples

Building off of the pattern established in Chapter 2 (§2.9), we provide examples of using our supportive tooling to encode functionality for analysis in McFIL.

Python3 DSL example for Yao's Millionaires:

```
def func_smt(solver, chosen_input, target_input):
    # Millionaire's functionality inside the solver
    return solver._if(solver._ugt(chosen_input, target_input),
                      solver.bvconst(1,OUTCOME_LEN),
                      solver.bvconst(0,OUTCOME_LEN))
```

```
def func(chosen_input, target_input):
    # Millionaire's functionality outside the solver
    return chosen_input > target_input
```

Python3 DSL example for Dual Execution (affine predicates):

```
def func_smt(solver, chosen_input, target_input):
    # isolate adversary-chosen function (matrix) from adversary-chosen input
    matrix_bits = solver.extract(chosen_input, CHOSEN_LEN-1, TARGET_LEN)
    chosen_input = solver.extract(chosen_input, TARGET_LEN-1, 0)
    # unpack matrix values
    matrix = [[solver.extract(matrix_bits, j*TARGET_LEN+i, j*TARGET_LEN+i)
              for i in range(TARGET_LEN)] for j in range(TARGET_LEN)]
    chosen_bits = [solver.extract(chosen_input, i, i) for i in range(TARGET_LEN)]
    target_bits = [solver.extract(target_input, i, i) for i in range(TARGET_LEN)]
    # perform mults
    chosen_matrix = [reduce(lambda x, y: solver._add(x, y),
                            [solver._mult(matrix[j][i], chosen_bits[i])
                             for i in range(TARGET_LEN)])
                     for j in range(TARGET_LEN)]
    target_matrix = [reduce(lambda x, y: solver._add(x, y),
                            [solver._mult(matrix[j][i], target_bits[i])
                             for i in range(TARGET_LEN)])
                     for j in range(TARGET_LEN)]
    # re-pack matrices
    chosen_out = solver.concat(*reversed(chosen_matrix))
    target_out = solver.concat(*reversed(target_matrix))
    # evaluate equality check
    return solver._if(solver._eq(chosen_out, target_out),
                      solver.bvconst(1,1),
                      solver.bvconst(0,1))

def func(chosen_input, target_input):
    matrix_bits = chosen_input >> TARGET_LEN
    chosen_input = chosen_input & ((1 << TARGET_LEN)-1)
    matrix = [[(matrix_bits >> (i+TARGET_LEN*j)) & 1 for i in range(TARGET_LEN)]
             for j in range(TARGET_LEN)]
    chosen_bits = [(chosen_input >> i) & 1 for i in range(TARGET_LEN)]
    target_bits = [(target_input >> i) & 1 for i in range(TARGET_LEN)]
    chosen_matrix = [sum([(matrix[j][i]*chosen_bits[i])%2 for i in range(TARGET_LEN)])
                        % 2
                     for j in range(TARGET_LEN)]
    target_matrix = [sum([(matrix[j][i]*target_bits[i])%2 for i in range(TARGET_LEN)])
                        % 2
                     for j in range(TARGET_LEN)]
    chosen_out = 0
    for i, bit in enumerate(chosen_matrix):
        chosen_out |= bit << i
    target_out = 0
    for i, bit in enumerate(target_matrix):
        target_out |= bit << i
    return 1 if chosen_out == target_out else 0
```

3.6.1 Evaluation Results

In this section, we evaluate McFIL by measuring its effectiveness in uncovering private `target` input(s) in the selected functionalities. Table 3.3 provides a summary of evaluation from repeated trials with random `target` values.

In each case, we include a graph denoting the decrease per query in remaining candidate solutions for `target`. These measurements are generated by ApproxMC, the approximate model counter of Soos *et al.* [SM19], over the course of a multi-run (representing multiple MPC executions, one per query) attack.

Process-Level Parallelism. McFIL uses parallelism in two key subroutines, one being `SelectOutcomes` and the other in computing the `chosen` query at each iteration. For an n-bit `target`, `SelectOutcomes` spawns one process per possible outcome (up to 2^n), and computing the query replaces an $O(n)$ linear parameter sweep from Delphinium with n parallel solving instances. For details on hardware, refer to §3.7.

3.6.1.1 Millionaires Problem

As previously discussed, the Yao's Millionaires functionality consists of a comparison between the adversary and hidden inputs. This enables a binary search which, as seen in Figure 3.5, eliminates approximately half of the remaining candidates at each iteration. The linear descent of the graph on a logarithmic scale captures the efficiency of the attack, resulting in approximately $log(n)$ queries for an n-bit `target`.

Parallelism in the attack generation pipeline achieves an approximately $2\times$ speedup in the Millionaires functionality. Savings occur in the parallel ApproxMC of the two outcomes in the `SelectOutcomes` heuristic. In this case, however, the heuristic is not required as the comparison predicate is complete and the two outcomes are not mutually exclusive. As a result, McFIL incurs additional overhead in exchange for its ability to handle mutually-exclusive outcome classes. We have added software arguments to configure (optimize) McFIL when the functionality is known to be complete.

3.6.1.2 Dual Execution

In this attack, McFIL generates a sequence of matrices and inputs in order to rule out candidate `target` inputs. Effectively, this attack creates an increasingly large system of linear equations choosing the coefficients and half the unknowns (`chosen`) at each step.

Figure 3.6 demonstrates the approximately logarithmic decreases of the remaining `target` search space over the course of an iterative attack. The plateaus visible in the graph further support the idea that there may be points at which McFIL is effectively "guessing and checking" within the available information, and once these guesses succeed (after 2-3 attempts, in the visualized attack) the following queries are able to eliminate a large quantity of candidate solutions.

3.6.1.3 Danish Sugar Beets Auction

In the Danish Sugar Beets Auction functionality, not all `target` bits are uncovered. Figure 3.7a in §3.7 denotes the rapid descent of remaining candidate solutions on a log scale, however, the attack does not reach $2^0 = 1$ (a unique solution). This seems to result purely from insufficient leakage (or, sufficient privacy) of the functionality. At the extremes of low and high prices, McFIL is able to determine the honest buyers/sellers bids for numbers of units. However, at middle prices closer to the likely equilibrium Market Clearing Price, McFIL fails to distinguish and some bits remain unknown.

In this example, we configured the adversary to control 3 buyers and 3 sellers, and the honest party to control 1 buyer and 1 seller. The adversary's goal (and therefore McFIL's `target`) was to uncover the prices and numbers of units of the honest

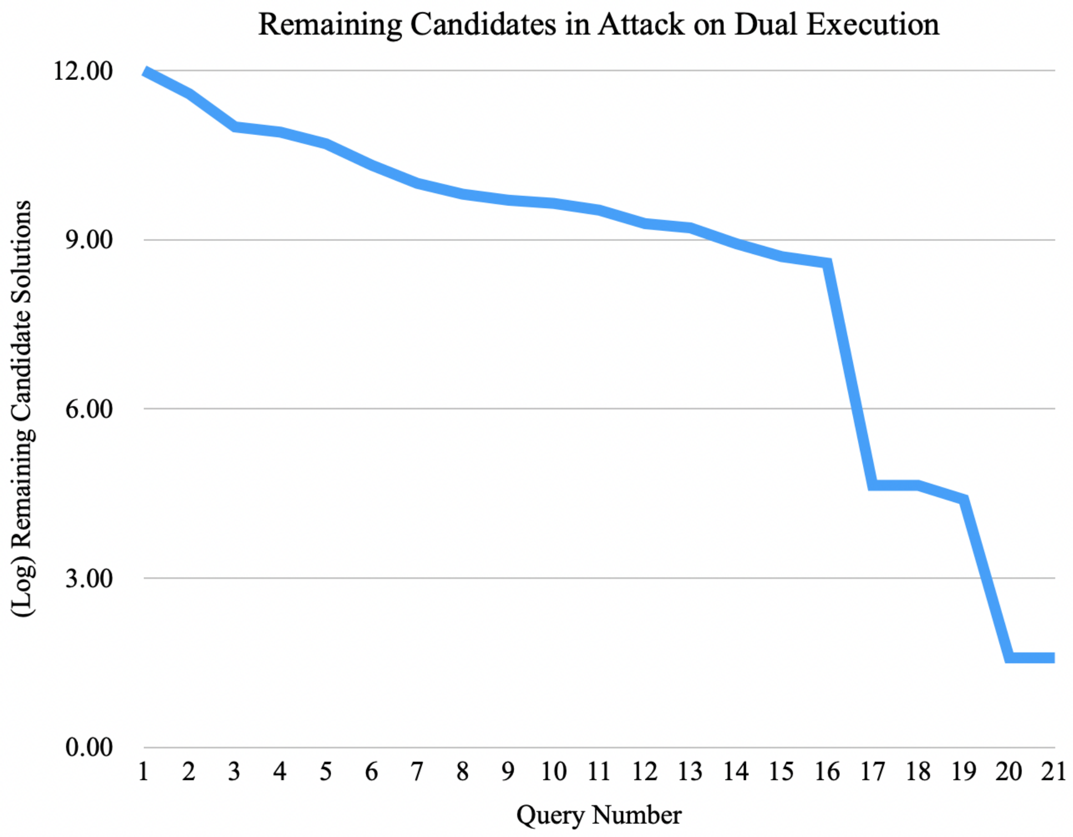


Figure 3.6: Remaining candidate solutions per query in a single trial of Dual Execution (Affine Predicates)

parties' orders. Although only partial knowledge is attained, the security model of the MPC [Bog+09] treats the entire prices/amounts list as confidential, and so this attack still represents a meaningful compromise of the intended privacy. Additional parameters and configurations of malicious and honest parties are explored in §3.7.

3.6.1.4 BWWC

We evaluate three functionalities derived from the Boston Women's Workforce Council MPC [Lap+16]: Bucketed Average, Wage Equity (with two variants), and Mean Average.

Bucketed Average. In the Bucketed Average functionality, McFIL is only able to uncover the `target` to the granularity of the buckets (ranges). By definition of the functionality, averages which land in the same buckets are indistinguishable, and so this result is expected. Figure 3.7b in §3.7 demonstrates the progress of the iterative attack, noting that it completes before finding a unique solution for `target`.

Due to the significant number of outcome classes, the `SelectOutcomes` subroutine dominates execution time. Specifically, the CNF generation step prior to parallelization incurs the most overhead. We discuss the overhead of CNF generation in §3.8. Once CNFs have been exported, parallelized ApproxMC may be employed, and due to the significant number of independent ApproxMC tasks the speedup in this case achieves up to 10× at some iterations.

Wage Equity. In both the *circuit division* and *standard division* variants of the functionality, the wage-average computation leaks significant information of the high-order bits of the honest party's input salary. Although the salary is not uncovered in full, learning the most significant upper half of this value effectively defines the salary range – disclosing information intended to be private.

In the *circuit division* instance, the mean calculation is reorganized to use a multiplication inside the solver rather than an unsigned integer division. Integer division corresponds with a complex Boolean formula after translation through Z3's SMT interface into SAT due to the edge-case handling of division. By applying some basic mathematical insight to adapt the target functionality to be more amenable to SAT, we achieve a notable increase in performance.

The performance gain is evident compared to *standard division*. As noted in Table 3.3, the overall computation time at a given bit width differs by an order of magnitude. However, as indicated by the similarity of Figure 3.7c and Figure 3.7d in §3.7, the resulting attacks achieve similar *per-query* efficiency in eliminating candidate solutions.

Both functionalities encode predicates with two simultaneously satisfiable outcomes. As a result, the `SelectOutcomes` heuristic is unneeded, and as a result is pure overhead lost in exchange for the assurance that the attack will proceed even if the outcomes are or become mutually incompatible. However, parallelism minimizes the impact of this overhead attaining a $1.5-2\times$ speedup across the two outcomes.

Mean Average. In the Mean Average functionality, the output range of the functionality is the full domain of its inputs. This functionality takes two inputs and computes their mean. As the inputs are bitvectors rather than purely mathematical integers or reals, some complexity is introduced to this averaging through floored division and the $2^n - 1$ bound of an n-bit value. As a result, McFIL finds attacks which generally require two queries, occasionally three.

Although this relatively simple functionality admits a query-efficient attack, it is a useful benchmark due to the maximal number of possible outputs. The full-domain output (of exponential size in n) taxes `SelectOutcomes` to the maximum degree per `target` bit. As a result, the bottleneck of McFIL is the `SelectOutcomes` heuristic, specifically in CNF generation for each SAT instance. CNF generation via the Tseitin transformation [Tse68] occurs within the Z3 SMT solver and, while asymptotically efficient, can require significant computation time. Unfortunately, as CNF generation vastly exceeds ApproxMC solving time within `SelectOutcomes`, process-parallelism offers little benefit and even occurs slight overhead in some tests.

Mean Average-like functionalities are the key motivator for the `SelectOutcomes` heuristic. With a Delphinium-like approach, the attack immediately fails with an UNSAT result. The reason for this failure becomes clear by example.

Consider a 2-bit adversary input a, and 2-bit honest input b. $\texttt{mean}(a,b) \in \{0,1,2,3\}$. Configuring simultaneous maximization to find an optimal a, the following constraint (among others) is added to the solver. The symbolic representations of the four partitions of b denoted $\{b_0, b_1, b_2, b_3\}$ correspond to b when the mean with a is 0, 1, 2, and 3, respectively.

$$mean(a, b_0) = 0 \land mean(a, b_1) = 1 \land$$

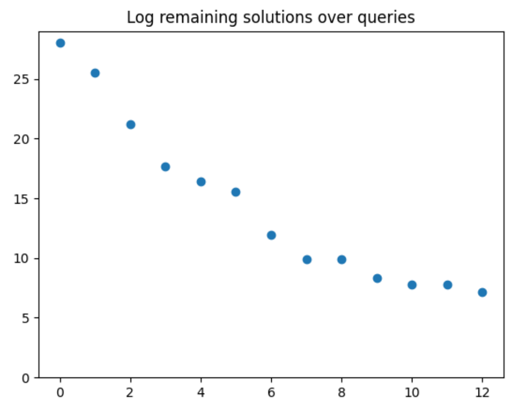
$$mean(a, b_2) = 2 \land mean(a, b_3) = 3$$

Notice that the first clause of the conjunction requires (a, b_0) to be $(0,0)$, $(0,1)$, or $(1,0)$ (with floored division). Therefore $a \in \{0,1\}$. However, the last clause requires $(a, b_3) = (3,3)$ exclusively. As $\{0,1\} \cap \{3\} = \emptyset$, the conjunction is unsatisfiable. `SelectOutcomes` correctly identifies that b_1 and b_2 are not similarly mutually exclusive, and that they contain the largest number of candidate solutions for b. As a result, b_0 and b_3 are selected out, and the attack proceeds without interruption. McFIL is then able to derive an optimal or near-optimal attack in 2 – 3 queries.

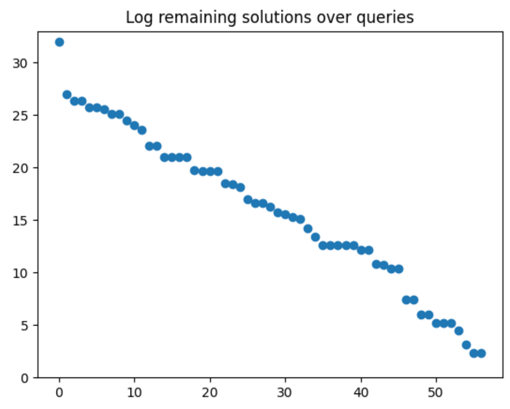
3.7 Additional Evaluation

Each graph within Figure 3.7 below denotes a representative example attack generated by McFIL. Each test operates over a domain of 28- to 36-bit values in order to keep overall benchmarking time reasonable with many repetitions of each attack. Some attacks do not trend to 2^0 as the functionality does not admit complete leakage of the underlying secret (honest) input(s).

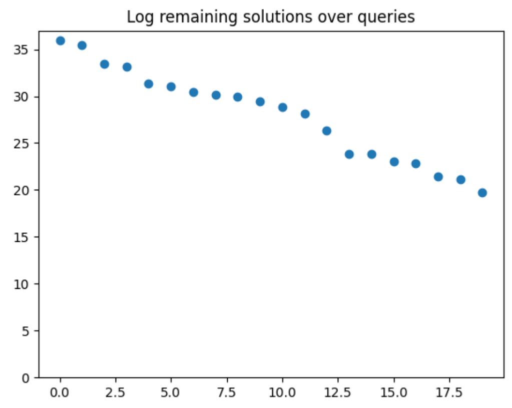
Testing Environment. All tests were performed on an Intel Xeon E5 CPU at 2.10GHz with 500GB RAM. Process-level parallelism was configured to employ 64 of the available virtual threads of execution using a Python3 process pool [Pyt22]. We used CryptoMiniSat 5.8.0, ApproxMC 4.0.1, Z3 4.8.15, and Python3 3.8.10.



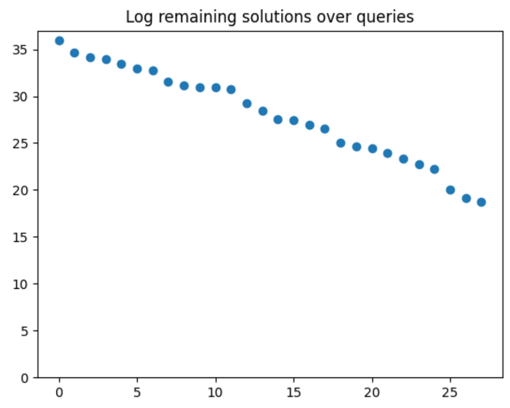
(a) Danish Sugar Beets Auction (2^{28})



(b) BWWC Bucketed Mean (2^{32})



(c) BWWC Wage (circuit division) (2^{36})



(d) BWWC Wage (standard division) (2^{36})

Figure 3.7: Remaining candidate solutions (log scale) per query

Many-Party MPC in the Sugar Beets Auction. The Sugar Beets Auction functionality demonstrates McFIL in the setting where more than two MPC participants are involved. In effect, McFIL treats multi-party protocols as 2PC: the solver is simply aware of bits that are symbolic and out of its control, and bits that it is able to vary to induce satisfiability. However, as it is an illustrative example, we demonstrate the varying results of McFIL as the number of honest and colluding malicious parties varies. Often, the uncovered `target` bits does not completely cover the domain, but in these cases McFIL generally uncovers one or more high bits of each Honest party's

Table 3.4: Additional Sugar Beets Auction Evaluation

	`target`	**Domain**	**Queries**	**Time**	**Speedup**
4 Players, Sellers Malicious	14 – 18	2^{28}	11 – 15	≈ 5m	4 – 5×
4 P., Buyers Malicious	8 – 18	2^{28}	8 – 15	≈ 5m	4 – 5×
4 P., One Each Mal.	12 – 12	2^{28}	11 – 18	≈ 5m	4 – 5×
4 P., One Buyer Mal.	0	2^{42}	6 – 10	≈ 3m	4 – 5×
4 P., Three Buyers Mal.	14	2^{14}	8 – 9	≈ 1m	2×
6 P., 2 Mal. 1 Hon. Each	10 – 14	2^{28}	11 – 12	≈ 5m	2 – 4×
6 P., 1 Mal. 2 Hon. Each	8 – 20	2^{56}	12 – 20	≈ 25m	1 – 7×

`target` bits discovered (maximum $log_2(domain)$)

Average time given for full attack, not per-query.
Minimum and maximum queries listed.
‘Speedup’ denotes approximate wall clock time savings through parallelization.

input, restricting their possible values to smaller ranges within the domain.

3.7.1 Contributed Benchmarks

At each iteration of McFIL execution, the solver contains a representation of all input knowledge. This includes the functionality and basic constraints which define the problem. In addition, it contains all iteratively added constraints which narrow the `target` search space. As a result, the McFIL procedure generates numerous SAT instances consisting of constraints directly relevant to the iterative discovery of leakage. This represents a relatively novel domain of usage for SAT.

In order to promote the development of SAT solvers and other tools, the SAT research community aggregates and shares “benchmark” SAT instances (generally stored as CNF or SMT2 text file formats). The SMT-LIB benchmark suite [BFT16] accepts submissions of contributed SAT instances in these text-based formats. The newest benchmarks in SMT-LIB inform the development of solvers, and provide an opportunity for SAT research to hone its tools on problems of practical interest. Historically this has aided solver development and improved performance on instances in the same class of functionalities over time.

As an additional artifact, we have submitted a wide assortment of benchmarks to the SMT-LIB compendium of SAT benchmarks [BFT16]. These have been accepted to help aid solver development and increase performance on the particular instance types McFIL encounters. Advancements in SAT research can offer a drop-in improvement to this work by expanding the horizon of computational feasibility. A summary of the submitted files is presented in §3.7.1. A subset of these benchmarks have already begun to see use within SAT research as a basis to test a new model counting approach [Sha].

Finally, in the course of generating, gathering, and testing numerous randomized SAT instances, a variety of underlying software bugs in the CryptoMinisat [SNC09] and Z3 [MB08] solvers were uncovered, reported to open-source software maintainers, and resolved collaboratively.

Table 3.5: CNF Benchmarks Overview

	#	**Clauses**	**Avg. Cls.**	**Variables**	**Avg. Var.**
Mil.	1,504	1,158 – 62,676	17,116	506 – 17,039	4,524
Dual Exec.	729	4,898 – 49,160	8,562	1,549 – 12,375	2,290
Sugar Beets	88	49,020 – 159,349	112,178	10,451 – 27,640	19,834
B. Mean	345	29,794 – 66,138	46,238	8,824 – 17,036	12,593
Wage	210	21,898 – 94,822	75,813	4,987 – 12,208	9,072
Wage (c.d.)	194	6,058 – 11,053	7,655	1,283 – 2,465	1,571
Mean Avg.	28	1,646 – 76,440	49,966	503 – 19,682	13,408
Total	3,098				

3.8 Discussion

We conclude this chapter with retrospective analysis of McFIL's practicality in our intended use cases, limitations, and then looking forward to additional open problems uncovered throughout this work.

3.8.1 Practicality

Our results demonstrate that McFIL is capable of evaluating a diverse array of simple functionalities in a matter of minutes of computation time. The generated attacks, which rely on the aforementioned multi-run adaptive assumption, are able to uncover all bits of the private `target` value in many cases. When our approach cannot continue to differentiate classes of candidate solutions to eliminate, it notifies the user that the attack may devolve to "brute-force" and asks if they'd like to continue. This occurs when `SelectOutcomes` fails to find a simultaneously satisfiable set of outcomes.

We observe that predicate functionalities execute relatively quickly, and functionalities with larger outputs (e.g. the Danish sugar beets auction and bucketed mean functionalities) spend significant time in `SelectOutcomes`. On the extreme end, the mean average functionality has an output domain as large as the output bitvector allows – and we correspondingly observe the most time spent in navigating these highly mutually-exclusive outcome classes.

In multiple cases, the `SelectOutcomes` heuristic is the computational bottleneck for attack progression. Analyzing these cases more closely reveals that the CNF generation step, using Tseitin's method [Tse68] to generate a CNF from an arbitrary Boolean formula with only linear expansion, is the crux of the subroutine. Despite its asymptotic efficiency, this step (as executed within Z3) takes significant time. We employ a number of CNF manipulation and caching techniques to avoid unnecessary regeneration of

CNFs where possible, however, the relative speed of parallelized ApproxMC leaves this sequentially-executed step as the longest-running component.

Even without the multi-run adaptive assumption to allow an iterative attack, the initial analysis step quantifies approximately how much leakage can be exploited with a given optimized query. On its own, this enables privacy/confidentiality analysis of any functionality planned for inclusion into an MPC, FHE, or ZK scheme. Further, this analysis needs to be run only once for a given functionality, and so even if hours or days of computation are required for some complex functionality, the resulting quantified leakage or iterative attack can provide pivotal insights to researchers and developers.

Limitations. The remaining practical limitations of McFIL are largely entangled with the inherent complexity of the computational problems it employs. For sufficiently complex functionalities or large output domains, running time increases significantly. Depending on the use-case, this may not rule out the use of McFIL for important/sensitive functionalities or contexts. Notably, for functionalities with very large output domains, or which encode cryptographic primitives (e.g. the AES round function [AKMY10]) directly within their Boolean circuitry, McFIL reaches wall-clock performance bottlenecks which may impede its applicability.

McFIL seeks to reduce operator burden and required expertise to develop confidentiality attacks or measure leakage. However, a certain degree of operator involvement is still required: McFIL can only be as accurate as the Boolean formula representation of a target functionality. The Python3 DSL we provide is an initial step in this direction, however it still requires understanding and manual effort.

3.8.2 Open Problems

SAT and cryptography have intersected numerous times in the research literature, often to their mutual benefit. McFIL pursues this interdisciplinary exchange by applying emerging SAT techniques to a new domain, bridging theory and practice, and contributing back to the SAT community. With McFIL, developers of secure protocols can automatically determine privacy thresholds or generate attacks against candidate systems. Potential users of these tools can evaluate them before choosing to use them. For sufficiently sensitive use cases, extensive computation times for complex or large functionalities may be worthwhile; McFIL empowers practitioners to make that decision without requiring expensive expert analysis. In addition to the opportunities to address limitations of this work, our approach offers a few promising avenues for future work.

Differential Privacy Thresholds. A differentially-private (DP) database protects individual data records (e.g. of people) while enabling aggregate queries. DP is generally parameterized by a number of allowable queries before privacy loss can be expected. Exchanges with a DP database can be viewed as a 2PC between the querying party and the database operator. The functionality executed is the aggregation function, and the privacy of individual records must be maintained. McFIL may be applied directly to verify the query threshold of DP systems. As an extension, McFIL could also be reconfigured with the intended threshold fixed to a concrete number of queries, and then used to solve for the privacy budget value. This would automate the process of choosing DP parameters, removing the need for expert intervention.

Evaluating Side Channels. Side channels appear as open problems in our discussion of Delphinium in Chapter 2, and remain open even given the contributions of this work. However, this chapter contributes a significant step forward in resolving this open problem by transitioning analysis capabilities from simple 1-bit predicate functions to functions with arbitrary finite output sizes. While a Delphinium-based approach would require the user to “bucket” function outputs to 0 or 1 in order to analyze, thus *deciding* the threshold as a parameter, a McFIL-based approach would allow a simple direct encoding of side channel measurements to bitvectors of a sufficient length, enabling the algorithm to *derive* any relevant thresholds dynamically and automatically. In our evaluations of McFIL we compare with the approaches of numerous works relating to side-channel leakage. Future work could entail reproducing or improving the results of those works using our approaches and then directly comparing an McFIL-powered analysis to further evaluate our approach and further elucidate the domain of leakage estimation.

Chapter 4

SocIoTy

This chapter is based on joint work with a number of contributing authors, originally published in the Privacy Enhancing Technologies Symposium 2024 [Joi+24].

Computing, ultimately, is a tool built by and for humanity. Equally as important as the types of defenses and attack, automation and estimation, or theoretical underpinnings thereof, is the translation of these concepts into their practical realization. In the modern era, the primary, most used, and most accessible manifestation of computing is the smartphone [Tos+12]. Therefore, in this chapter we naturally progress from automating attacks to developing usable defenses in the form of a secure protocol designed for home and smartphone use. This protocol and the accompanying analysis and evaluation of its usability and privacy constitute the primary contributions of the chapter. This work, resilient to the attacks we present in prior chapters, bridges theory and practice in its design and implementation.

In addition to being computing platforms that are deeply embedded in our daily lies, smartphones are important sources of trust, providing users secure access to their data everywhere they go. While this mobility is convenient, it also represents risk. If a user's smartphone is stolen or seized, an adversary can uncover or even impersonate the user in their digital life: accessing the user's data and services or decrypting their sensitive files. With this in mind, we design *SocIoTy*, which uses IoT devices in collaboration with a smartphone to provide a high-security trusted computing base for non-expert users. We first discuss the notion of *at-home cryptography*, which binds cryptographic operations to a user's physical presence in their home (or another pre-determined location). We then provide our instantiation of at-home cryptography by federating trust across the IoT devices that users already own. We implement and evaluate SocIoTy for practical, real-world use cases such as authentication and encryption. SocIoTy is able to provide strong cryptography while binding its execution to the IoT-capable home ("smart home") itself.

Mobile devices have quickly become the most important trusted computing base for most users. 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.

The convenience afforded by mobile computing is accompanied by a commensurate increase in risk. Mobile devices are easily lost; for example, if a user loses their mobile device used for two-factor authentication (2FA), they may be unable to access their bank account or their medical records. While account recovery may be possible, account recovery mechanisms are notoriously insecure and often exploited by attackers [AK12].

Worse still, if a mobile device is *taken*, the cryptographic material used to authenticate the user could be extracted [ZJG22]. This would allow the 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 anti-government.

Mitigating risk with at-home cryptography. 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. Users might already limit their use of online banking or telemedicine to times when they are at home for privacy reasons, for example.

For these applications, it would suffice to authenticate using cryptographic operations that could only be performed at home, a class of cryptographic operations we refer to as *at-home cryptography*. However, mobile devices offer no fine-grained access control mechanisms based on location that could facilitate at-home cryptography; even if the user only uses the material to authenticate themselves while at home, that material is still exposed regardless of location.

At first glance, achieving limited access to authentication material might appear trivial: a user can simply store authentication material 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 for additional security—access to the authentication material is inherently limited.

While straightforward, this solution requires two inconvenient preconditions. First, the user to *own* such stationary hardware (with a reasonable expectation that it will never be moved from the home), and second, the user must have the technical expertise to manage this hardware and any needed security configurations. 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% [Pew21].

At-home cryptography for everyone. In this work we study the feasibility of realizing practical, accessible at-home cryptography that can be widely used while remaining secure.

At-home cryptography is a special case of *location-based cryptography*, which has been studied in both the theoretical [Cha+09; Bro+17; Buh+11; Phu+19; PQ10; Kas+12] and applied [SD03; Qiu+07; FV12; AFAI07] literature. However, no practical constructions have been contributed in these works. Further, there are impossibility results that might rule out some "ideal" solutions [Cha+09]. As such, we attempt a pragmatic approach, and make the following observations.

First, we observe that to achieve at-home cryptography, users can leverage existing hardware. Although users are unlikely to have access to *dedicated* secure hardware for at-home cryptography, they may have access to Internet-of-Things (IoT) devices. These single-purpose, generally inexpensive devices typically do not leave the home, making them an attractive prospect for anchoring a trusted computing base for at-home cryptography. Second, we note the possibility to use this existing hardware to federate trust. Because IoT devices have an extensive history of vulnerabilities [SRB20; DV17], it is not advisable to use any single device as a trusted store of secrets (cryptographic or otherwise). The concept of *Defense in Depth* [Mug18] provides a guide: it is by definition always at least as difficult to exploit a set of multiple systems as it is to exploit any single system in that set. Thus, we observe that if we are to rely on IoT devices, it is more appropriate to distribute trust among *many* IoT devices, such that an attacker would need to compromise multiple devices before exposing any of the user's secret data.

Employing these observations to develop a practical system requires carefully navigating tradeoffs between security, usability, and efficiency. Because of the single-purpose nature of IoT devices, they generally have much lower computing capacity compared personal computers or even to smartphones. This in turn limits the complexity and flexibility of viable approaches to federating trust.

4.1 SocIoTy

In this work we provide a system design and related protocol for at-home cryptography using a user's existing IoT devices. We term this contribution *SocIoTy*. Our system is designed for non-expert users who wish to protect high-value digital resources but do not have the resources, expertise, or inclination to use dedicated secure hardware. Our system allows these users to set their own risk tolerances, allowing them to bind whatever data 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. We provide a summary visualization of our design in Figure 4.1.

SocIoTy achieves at-home cryptography by performing secure multi-party computation (MPC) [Yao86b; GMW87; BGW88; CCD88] using IoT devices which hold secret shares of cryptographic data such as encryption keys or authentication secrets. The

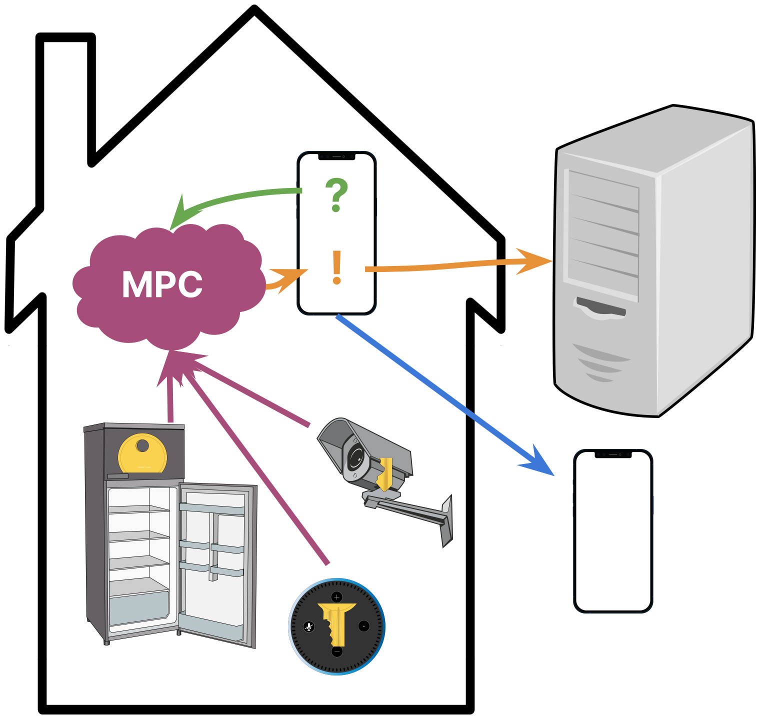


Figure 4.1: System architecture of SocIoTy. An authorized device interacts with the smart home to perform at-home cryptographic operations for interaction with remote services.

user, through their smartphone, initiates an at-home cryptography request by communicating with the IoT devices, which use short-range protocols such as Wi-Fi, Bluetooth, or Zigbee [AS+17]. The devices then collaboratively perform MPC to compute the output of the requested operation, returning it to the user via the smartphone. For example, in the encryption case, the operation can return a decryption key to the user, which they must simply securely erase after using; secure erasure is a standard feature on modern smartphones [ZJG22]. In the authentication case, the smart home can for example generate a one-time passcode, allowing the user can authenticate to a remote service once. The capability of MPC to support arbitrary functionality ensures that the range of available operations is limited only by the computing capability of the IoT devices, and the security of the MPC protects sensitive data from network interception or even a subset of compromised participant devices. The limited transmission power of IoT devices ensures the operation can only be performed within (or perhaps very near to) the smart home.

IoT devices are resource-constrained and will generally serve some primary purpose in the home beyond SocIoTy. Thus, they may need to stop participating in the MPC abruptly to handle user requests and events. In the naïve case, the devices would then have to restart computation with a different set of participants, dramatically increasing system latency. So, we must design an MPC protocol that does not strictly abort when parties spontaneously go offline. To that end, we employ a *fluid* MPC protocol. *Fluidity*, a recently formalized MPC property [Cho+21], measures an MPC protocol's tolerance to churn among participants. It was initially envisioned to facilitate long-running MPC evaluations of complex functionalities with large, dynamic groups. While SocIoTy considers relatively small cryptographic circuits, fluidity nonetheless provides resilience and relative efficiency in the event of dropouts.

Our implementation of SocIoTy supports symmetric authentication and encryption that is interoperable with various real-world schemes. Together, these can power many practical needs, such as the generation of two-factor authentication (2FA) one-time passcodes (OTPs), and allowing users to store and retrieve sensitive, encrypted content—all without worrying that they will risk compromise of their credentials outside the home. Moreover, service providers (*i.e.*the targets of an authentication operation) would not have to change their systems to accommodate SocIoTy; users simply change how they store their own secrets. Our system is also sufficient to construct *self-revocable* encryption [Tya+18], allowing users to temporarily delete key material that could decrypt sensitive data, for example stored in the cloud, and allow key recovery after the user has safely returned home. We find that our implementation meets the performance needs of our intended applications—performing OTP generation or key encryption, on average, in less than 3 seconds.

Contributions. In this chapter, we discuss the problem of giving non-expert users location-based access control to cryptographic material, focusing on the at-home cryptography setting. Our goal is to help average users mitigate the risk associated with carrying high-value cryptographic secrets or on their smartphones, giving users the peace-of-mind in knowing that their configured services or data can only be accessed from home. Specifically, we discuss *at-home cryptography*, and show how it can be used to build relevant constructions such as time-based one-time passcodes [M'R+11] and self-revocable encryption [Tya+18] that are restricted to the smart home. We then present SocIoTy, covering its system design, implementation, and evaluation. Further, we develop a domain-specific language for writing MPC circuits, and a fully-realized implementation of a generic MPC system in Rust with support for churn, as well as additional extensions to support SocIoTy.

4.2 Background

Smart home. SocIoTy relies on a network of Internet-of-Things devices within the home to bootstrap at-home cryptography. IoT devices and smart home networks are proliferating rapidly [Mor22]. In 2022, 57.5 million Americans lived in a smart home, accounting for 45% of United States households, and it is estimated that by 2026 more than 25% of homes worldwide will have some degree of IoT capabilities [Sta22]. IoT devices range in computational hardware from extremely lightweight micro-controllers to fully-Linux-capable systems with gigabytes of RAM. We rely on IoT devices to perform cryptographic operations and to communicate over the network to realize SocIoTy within an otherwise independent set of devices. As in prior work [Kum+19] we assume a network of constrained but somewhat capable devices (in terms of computation and power) rather than the bare minimums of computational capacity. In our evaluation

(§4.5) we model these devices with Raspberry Pis, lightweight and inexpensive but Linux-capable devices, to explore feasibility.

Location-based cryptography. Location-based cryptography was introduced formally by Chandran et al. in 2009 [Cha+09] 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. However, the authors also develop collusion-secure position verification by adopting a further assumption: in the bounded-retrieval model, provers are only capable of extracting partial data from high-entropy messages transmitted through the space. Using this tool, the authors demonstrate collusion-resilient location verification, and further develop a key exchange mechanism with location as its basis. Works since have explored the assumptions made by Chandran and their implications in complexity theory [Bro+17] and in the quantum setting [Buh+11]. Location-based cryptography has also arisen in rudimentary forms in industry and personal device use. Examples include hardware tokens which are kept physically secure (e.g. within a home or bank deposit box) or hardware security modules deployed to remote corporate offices – therefore requiring participation from that geographic location to authenticate an action.

Two-factor authentication (2FA). To increase security for end-users, online service providers have increasingly begun to deploy 2FA, which requires a second form of authentication to login to a service. The most common form of 2FA, after email and SMS [Che21], is the time-based one-time password (TOTP) [M'R+11]. Every x seconds (recommended default $x = 30$), a user's token (*e.g.*, a smartphone app) generates an OTP. When the user wishes to authenticate, they input their username, password, and the current TOTP code. Unlike passwords, TOTPs are short lived; they are only valid for the time interval x in which they are generated, and can only be used once. Users therefore authenticate with a combination of *something they know* (a password), *something they are* (a fingerprint or retina scan), and *something they have* (their token, which generates OTPs). TOTP is supported by major social media platforms [Wat21], electronic health records systems [Duo21], financial institutions [Rob], and corporations [Cis20].

The security of TOTP relies on the underlying HMAC-based OTP algorithm [M'R+05] (HOTP), which generates OTPs using the HMAC construction [KBC97]. The security of HOTP, in turn, relies on the assumption that HMAC is a pseudorandom function (PRF) [GGM86]. A PRF is a function with outputs that cannot be distinguished from a random function without some secret key sk. Since adversaries without sk cannot predict PRF outputs, PRFs are useful for generating OTPs. Thus, as long as we assume that our underlying primitive (HMAC) is a PRF, then the OTPs generated by HOTP (and thus TOTP) are secure. More formally, a TOTP is parameterized by x (seconds) and defined by $\mathsf{TOTP}_x(sk,ts) = PRF(sk, \lfloor \frac{ts}{x} \rfloor) \mod 10^6$, where ts is a timestamp and $\mod$ is the modulus operation (used to convert the output of PRFinto a 6-digit integer). We omit x in our notation for TOTP for simplicity.

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 [ZJG22], as well as defending cryptographic protocols from compelled decryption by identifying and reducing long-lived secret values [SV21]. BurnBox [Tya+18] attempts to address compelled decryption by putting a users ability to decrypt in escrow in a secure location—specifically by allowing secure deletion which is recoverable 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 [Cha+09; SD03; Phu+19]. 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.

Secure multi-party computation (MPC). As described in Chapter 3, MPC, which was first described in early works of modern cryptography [Yao86b; GMW87; BGW88; CCD88] and has since been extensively explored and developed, allows a set of parties to compute a function of their inputs while keeping those inputs confidential. A generic MPC protocol Π can securely compute some arbitrary function f, described as either a Boolean or arithmetic circuit. However, the security and properties provided by Π may differ widely. Some protocols may provide security against either a majority or minority of adversary-controlled parties, or against an eavesdropping or fully-malicious adversary. They may also provide other properties, like guaranteeing protocol output even when parties deviate from the protocol (termed *guaranteed output delivery*).

Recent work has studied the setting of *dynamic participation*, in which the parties that participate in the protocol may change over time. In this work, we follow prior work [Cho+21] where f is expected to be a long-running computation—such as training a machine learning algorithm—where it may be unrealistic to expect parties to participate for the duration of the whole protocol. We adapt the BGW protocol [BGW88;

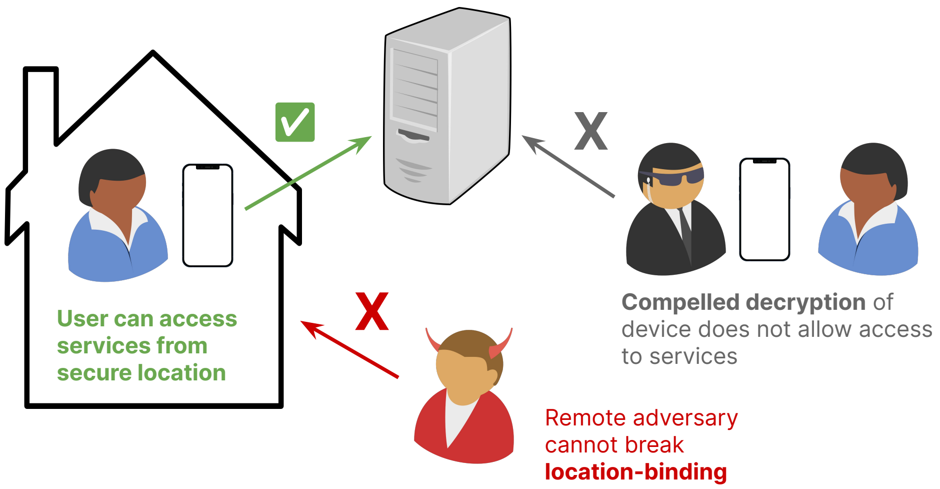

✅
X
User can access services from secure location
X
Compelled decryption of device does not allow access to services
Remote adversary cannot break location-binding

Figure 4.2: Overview of the SocIoTy threat model.

GRR98] to this setting and show that malicious security may also be achieved by using a variant of the malicious compiler of [Chi+18]. Other works have followed the idea of dynamic participation models in MPC with various trade-offs between network performance, computational complexity, and resilience [RS21; Gen+21; KRY22; Cas+22].

4.3 Threat Model

In this work, we consider a threat model which relates directly to most end-users of home IoT devices. In our setting, we consider a physically secure home location to which the user has consistent access. This serves as a trusted location where decryption or other sensitive actions may be allowed to occur. Further, our model requires two classes of devices to be present. First, the user has a single *authorized device* (*e.g.*, a smartphone or laptop) which moves with them and is not always home. This device may be lost, stolen, or seized; in any of these cases we assume that all of the devices secrets can be extracted (either through circumventing on-device protections or through compelled decryption) [ZJG22]. We otherwise assume the device to be secure and the software it runs to be trusted.

Second, our system leverages a number (say, N) of IoT devices provisioned within the home, which communicate over a shared, wired or short-distance wireless (*e.g.*, Wi-Fi) network. We focus in particular on devices that are difficult to remove from the home, such as large smart appliances, lights, hubs, and security cameras. Of these N devices, $n \leq N$ of them will be participating in the at-home cryptography protocol at any point in time. We assume that some threshold t of the devices that are participating in the protocol can be compromised, allowing the adversary total and arbitrary control over them. In our implementation we will make an *honest-majority assumption*, *i.e.*, if n of the N total devices are participating in the protocol at any given time, security is maintained when $2t < n$. We argue that, for small numbers of devices within the home, it is reasonable to assume that up to only one or two can expected to be actively compromised at any given time. The user may be unaware that a compromise has occurred, and the adversary is assumed to be able to use the compromised devices to interact with the rest of the network any way they choose. That is, we essentially assume a fully privileged remote-code-execution exploit.

Security goals. An overview of our threat model can be found in 4.2. Our work addresses adversaries who wish to break the *location-binding* property. That is, the constraint that only users possessing both the authorized device *and* who are physically at home can successfully execute the protocol. The smart home devices provide federated trust and an effective “proof of location” that our system will leverage, and thus are the natural target of the presumed adversary’s attacks. For example, an adversary could use a compromised IoT device to attempt to masquerade as the authorized device, or otherwise corrupt the protocol to perform sensitive actions without the user present at home.

While all adversaries we consider have the ability to compromise t devices (*i.e.*, a minority fraction of the network), we also consider adversaries with two additional capabilities. First, we consider adversaries that can physically access the home and may try to interact with the at-home cryptography system, *e.g.*, a malicious house guest. These adversaries can eavesdrop on the local network and send messages to the IoT devices. They can also interact with the user's authorized device, but cannot compromise it.

Second, we consider adversaries that can obtain access to the authorized device when it is not physically present in the home. These adversaries can extract all of the secrets from the authorized device and use this in conjunction with any IoT devices in the home they may have compromised. For example, consider a border control officer that compels decryption of the user's authorized device while the user is crossing the border. They may also have been able to compromise some of the user's IoT devices remotely, but are assumed to not be near the home location. We note that this threat model exceeds that of many secure protocols, which assume a malicious network but an unquestionably trusted, secure end-user device. Some works have considered this aspect of our threat model such as analyzing security against compelled decryption [SV21].

We assume that all of the user's devices (authorized and IoT) have access to point-point secure channels for communication. We can achieve this through the use of transport-layer security (TLS) with client-side authentication [Res18] (also referred to as *TLS mutual authentication*. In practice, all communications between devices will generally be restricted to the wired or wireless LAN. To achieve this, there needs to be an initialization phase where the user is able to securely provision all of the devices in the system—a normal step in most home IoT configurations. Part of this initialization will establish the necessary secrets for secure point-to-point channels between the devices. While this can be seen as a strong assumption, we observe that the user controls all of the devices involved, and the entire protocol execution occurs within the home.

Execution of our protocol begins by having the authorized device initiate. This initiation can only be triggered by the authorized device while it is at home (as otherwise it simply cannot communicate with the IoT network). The authorized device will be provisioned with the necessary credentials to carry this out during the initialization. The IoT devices will start execution upon receiving the initialization message from the authorized device. It is important to note that compromised IoT devices will be able to "listen in" to the computation and participate, but do not have the credentials needed to initiate the protocol. We discuss this in more detail later in this section.

Thus, we can summarize the security properties of SocIoTy as follows. Given:

(1) an initial provisioning phase to establish point-to-point channels,

(2) an adversary that can compromise no more than t IoT devices on the network, and

(3) an authorized device that may be lost or stolen, but will not be compromised while at home,

SocIoTy guarantees that there is no computationally-bounded adversary that can break the *location-binding* property.

Network exposure and device cloning. The majority of IoT devices communicate with the external Internet, generally the manufacturers cloud servers. This represents additional attack surface, as an adversary could attempt to break location-binding by compromising the authorized device outside of the home, compromising a single IoT device in the smart home, and forwarding authorized commands from the device to the other IoT devices. To mitigate against this, we establish a "wake on LAN" configuration wherein a non-Internet-connected device receives authorization from the smartphone over a low-range physical layer, which then activates the devices actually involved in the computation. Many common IoT devices communicate only via low-range radio (*e.g.*, NFC [ECM13; Nor]), obviating the need for additional hardware in this step.

To limit the need to perform this low-range activation process, we can utilize a short range key recovery procedure. First during initialization, each IoT device gets a share of the authorized device's authentication credentials (this is natively supported by the at-home cryptography system). Then, prior to compelled decryption of the authorized device, the credentials can be deleted from the device (this is equivalent to the BurnBox [Tya+18] setting). Once the user returns home, they use the NFC-based "wake on LAN" procedure to reconstruct the authorized devices credentials. Following this key recovery, the at-home cryptography system will work as normal again.

We note that while this "wake on LAN" assumption may seem to introduce an effectively "air-gapped" device into our system, we observe some key differences which remain. Particularly, we still do not require specialized hardware outside what would commonly be found in a IoT-enabled household. Trust is still distributed, and no single device has access to the user's secret data. The flexibility of our system is also maintained, as the user only needs to make use of this process if they are concerned of a significant compromise of their state.

Security non-goals. The user (owner of the smart home) is considered to be honest. While a malicious user could break the location-binding property of our system, we consider this to be out of scope. The main focus of this system is preventing access by unauthorized individuals. We also assume that any remote service involved in an at-home authentication protocol is honest. Our primary concern lies with adversaries that can compromise the user's IoT devices or authorized device. Additionally, we allow the adversary to abort protocol execution, as this does not impact location-binding. Finally, we do not consider the threat of an adversary that is able to compromise the user's authorized device and *also* gain physical access to the home. Our protocol cannot defend against this as-is, because this represents total compromise of all of the user's protocol state.

4.4 At-Home Cryptography

To formally capture the notion that some cryptographic operations should only be available at home, it is necessary to "modify the interface" of notional cryptography function invocations (decrypt, generate OTP, etc.) with an additional input representing location. This modification clearly captures generic location-based cryptography, a superset of at-home cryptography. As we are only interested in the particular subset with location either in the home or outside of it. Thus, we modify a notional cryptographic function F with input z to produce the function F_{home} as follows:

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

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 to make queries to a subset of functionalities, where the subset was 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.

4.4.1 At-Home Cryptographic Applications

At the heart of any at-home cryptography application is an existing service that users want to restrict to a private environment in which they feel secure. By limiting access to these services to their home, users trade flexibility or convenience for additional security and privacy. However, we observe that there are many applications for which the loss in flexibility might be inconsequential or beneficial. Limiting the environment in which they can access a service might be an issue of comfort for some users, *e.g.*, accessing telemedicine services or banking services without worrying about shoulder surfers. For these users, the loss in flexibility might only marginally impact their lives, while the peace-of-mind offered by at-home cryptography might be significant. Other users might be interested in leveraging at-home cryptography to enforce cybersecurity best-practices, *e.g.*, a doctor's office or law practice allowing remote work but still controlling the locations from which employees can access the corporate VPN or decrypt highly sensitive documents. For these users, there should be no loss in flexibility, as at-home cryptography only impacts access patterns that are already forbidden by existing policies.

An inherent tradeoff. We emphasise that our intention is to allow users to *opt-in* to this extra layer of protection for the selected services that make sense for them (or organizations they belong to); critically, these choices are highly personal to each user. Users with chronic or acute illness, for instance, might require access to telemedicine services at all times and cannot afford the privacy-flexibility trade-off that location-based access provides. For other users, the choice to limit access to telemedicine services to their home might be very natural. Consider, for example, a user with an at-home dialysis machine, which 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 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, a user planning to cross international borders (where government agents generally have broader search powers) might want to add an additional layer of security to their sensitive services, even if they do not location-bind access to these services in their daily life.

4.4.2 At-Home Cryptographic Primitives.

We consider two cryptographic tasks that users might want to limit to their home: *at-home authentication* and *at-home encryption*. While there are a large number of other cryptographic tasks that one might want to consider limiting to the home, authentication and encryption are sufficient for most practically important applications.

At-home authentication. At-home authentication allows users to authenticate to services that they only want to be accessible from home. We imagine that these services will most likely be online portals, social media accounts, or corporate VPNs. When logging into these services, the user's home becomes a second factor—somewhere you *are*, in addition to the typical factors of something you know (passwords) or have (hardware tokens). The protection offered by at-home authentication will most likely be challenged by low-resource attackers who might try to impersonate the user after stealing their mobile device. High-resource attackers, on the other hand, might be able to breach the security measures on the online service itself, thereby circumventing authentication altogether.

To build at-home authentication from at-home cryptography, we select the cryptographic functions used in everyday authentication workflows to be executed within the at-home protocol, namely *at-home signatures* and *at-home OTPs* (looking ahead, we will implement and evaluate at-home OTPs). For example, FIDO [Bar+20] and client-authenticated TLS use signatures for authentication, and many online services support time-based OTP, usually managed by an on-device app, to provide users with their second authentication factor. These primitives can be replaced with their at-home analogues (at-home signatures and at-home pseudorandom functions, respectively) and then seamlessly used in the original workflows, achieving at-home authentication.

At-home encryption. At-home encryption allows users to secure files such that they can only be decrypted when the user is at home. This is analogous to storing physical files in a safe or lock-box in the home, as is common practice for financial documents, birth certificates, or documents with sentimental value. At-home encryption facilitates the creation of a digital equivalent to these. Importantly, it is possible to accomplish this task robustly by storing encrypted files in the cloud—even if the encrypted files are available globally, the decryption keys and therefore the corresponding plaintext data are location-bound.

We note that the notion of a "digital safe" was contemplated in BurnBox [Tya+18]. Tyagi et al. proposed the notion of self-revocable encrypted cloud storage to protect travelers against compelled decryption when crossing national borders. A digital at-home safe was a critical component for recovering revoked files after the threat has passed, and the user has returned home. The notion of at-home encryption captures this application, and our realization of at-home encryption allows even non-technical users to take advantage of the BurnBox system without purchasing dedicated hardware.

At-home encryption can be realized with either directly replacing the encryption and decryption with their at-home counterparts, or by designing *at-home key derivation function*. When using the key derivation function, the user can query the key derivation function on the identity of a particular files to recover the necessary key material to either encrypt or decrypt the files directly. After interacting with these files, the key material and the plaintext files can be securely deleted from the user's device.

4.5 SocIoTy

In this section, we describe aspects of our SocIoTy construction, and how they relate to the primitives of at-home cryptography discussed in 4.4. We discuss SocIoTy in terms of its constituent components:

– *Authorized device/authorized smartphone*: This device has moderate computational capacity and is carried by the user. We assume the 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 [ZJG22].

– *Remote service*: The remote service is an Internet-accessible service with which the user wishes to interact, leveraging 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. For performance, the user selects those devices with sufficient hardware and network capability to execute the protocol (*i.e.*, excluding microcontroller-class devices). Recall that that only a fraction of these devices may be compromised by the adversary.

For both at-home authentication and encryption, we need to bind at least one irreplaceable step of the operations to the particular location. One way to do this is to have one device in the home generate the key material necessary to do authentication and encryption, and execute all of the necessary algorithms when contacted by the smartphone. However, IoT devices are not sufficiently robust [SRB20; DV17] in terms of security or computational capacity for this to be acceptable. Compromising a single IoT device with weak security would lead to the compromise of location-binding and the whole system: a scenario which is at least plausible, if not likely.

To mitigate the impact of IoT device compromise, we evaluate the standard cryptographic functions used in authentication and encryption within an MPC protocol distributed amongst multiple devices. Specifically, the secret key material is secret shared among all N IoT devices, and only ever reconstructed *within* the MPC. Thus, compromise of one device—or even a handful of devices—does not invalidate the security guarantees of the system. Throughout this work, we use AES256 and MiMC [Alb+16] as our symmetric cryptography primitives. AES256 is useful because of its maturity and relative ease of implementation in arithmetic MPC protocols, and is a common benchmark for the efficiency of MPC protocols [Kel+17; DK10]. We also study MiMC because it is a relatively new MPC-friendly cipher with low multiplicative complexity; similar ciphers have been explored in [Alb+15; AD18; Aly+20; Gra+21]. For AES256, we represent our computation as an arithmetic circuit over $GF(2^8)$ whereas for MiMC we use $GF(2^{128})$.

We now describe in depth how both our primitives can be realized according to a generic MPC protocol Π that computes AES256 or MiMC, given shares of the key as input and a message, provided by a single party as private input.

IoT device registration. In the setup phase, the authorized device wishes to enroll a new IoT device into the smart home and the protocol. The authorized device prompts the smart device to create relevant key material, and the newly created public key information is disseminated to all other devices in the smart home by the authorized device. The new device is also given all public information of the other devices within the smart home.

4.5.1 At-Home Authentication

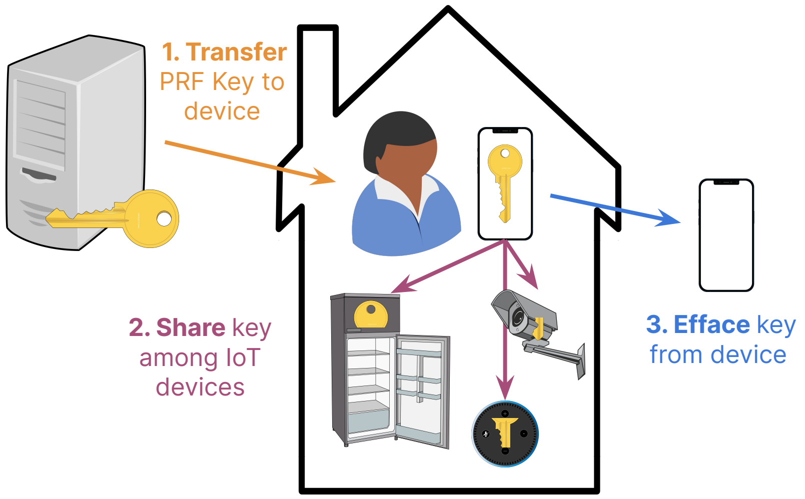
To build an at-home authentication scheme using TOTP, we construct $\mathsf{TOTP}_{\text{home}}$ as

$$\mathsf{TOTP}_{\text{home}}(ts, \mathsf{loc}) = \begin{cases} \mathsf{TOTP}(sk, ts) & \text{if } \mathsf{loc} = \text{home} \\ \perp & \text{otherwise} \end{cases}$$

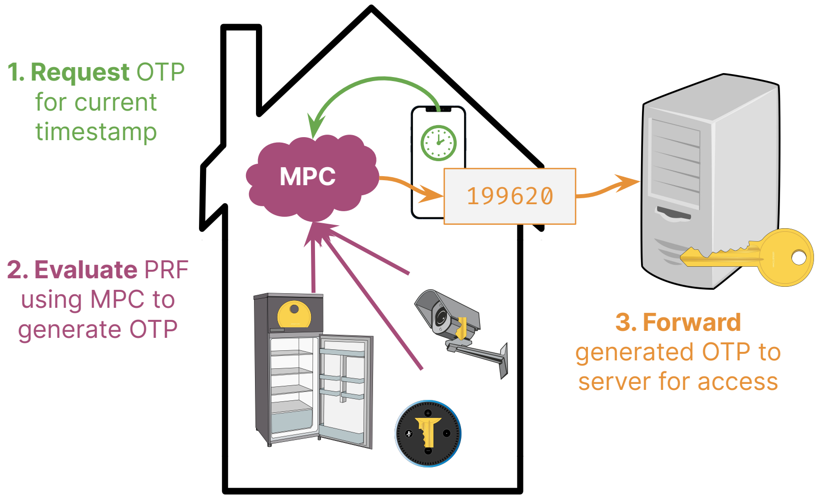
Note that secret key sk is not present as a parameter to $\mathsf{TOTP}_{\text{home}}$, as it is unavailable to the user. Instead, sk is bound to the location parameter, which means that it is only available if $\mathsf{loc} = \text{home}$. Two questions remain: how should we instantiate TOTP, and how do we generate codes from TOTP?

Instantiating TOTP**.** As discussed in 4.2, typical deployments use HMAC to instantiate the underlying pseudorandom function PRF of TOTP; indeed, the OTP RFCs explicitly assume that HMAC acts as a PRF in their security analyses [M'R+05; M'R+11]. Unfortunately, it is inefficient to evaluate HMAC within an MPC protocol, as it leverages SHA-256 as a subroutine. Even the optimized SHA-256 circuit created for SCALE-MAMBA has a multiplicative depth of over 1,500, and the gate-by-gate MPC protocols we consider have round complexity proportional to the multiplicative depth of the circuit being computed [Arc+].

We therefore propose replacing HMAC within TOTP protocol with functions that have a simpler circuit representation. Block ciphers are typically modeled as pseudorandom *permutations* (PRPs), but prior theoretical works [Hal+98; BI99; GGM18]


1. Transfer PRF Key to device
2. Share key among IoT devices
3. Efface key from device

(a) Initial setup: transfer, share, efface.


1. Request OTP for current timestamp
MPC
199620
2. Evaluate PRF using MPC to generate OTP
3. Forward generated OTP to server for access

(b) Protocol execution: request, evaluate, forward.

Figure 4.3: SocIoTy for at-home authentication.

have shown that truncating the output of a PRP results in a (secure) PRF. As defined [M'R+11], TOTP truncates the output of its PRF to be 6 digits long. Thus, as long as our PRP-to-PRF truncation results in enough bits to allow further truncation to 6 digits ($PRF \mod 10^6$), we can use block ciphers as the PRF in TOTP.

With this in mind, we set $PRF(sk,t) = \mathsf{Enc}(sk,t)[:24]$, where Enc is the symmetric key encryption function of a block cipher like $\mathsf{AES256}$ or MiMC, and the output is truncated to be 24 bits long. Since $2^{24} > 10^6$, truncating the output to 24 bits will still allow for OTPs that are at least 6 decimal digits, meeting the current interface for TOTP. Using only 24 bits of PRF output introduces slightly more bias: the first 777,216 possible 6-digit OTPs have a probability of occurring roughly equal to 1.013×10^{-6}, whereas the rest occur with probability 0.95×10^{-6}. However, given the ephemeral nature of TOTP, this bias is unlikely to be exploitable. Also, since ts is a Unix timestamp (the *time-based* component of TOTP), it is only 8 bytes long (modern Unix systems define `time_t` as a 64-bit integer to avoid the Y2K38 problem), which means it will fit within one block of an $\mathsf{AES256}$ or MiMC execution, making a block cipher mode of operation unnecessary (*i.e.*, we only need "ECB mode"). We can then use this PRF as a drop-in replacement for HMAC in TOTP.

Creating $\mathsf{TOTP}_{\mathbf{home}}$ from TOTP. Now that we have defined our TOTP and its PRF, we proceed to setting up $\mathsf{TOTP}_{\mathrm{home}}$, following the steps denoted in 4.3a. We assume that the user of the authorized device would like to use at-home authentication for connecting to a server that uses TOTP. To register a service, the authorized user first obtains a key sk to be used with the service while at home. This key is then secret shared as a (t,N) sharing to all N IoT devices that make up the smart home network. Then, sk is *effaced* (securely deleted) from the authorized device. This initial setup encodes the loc property of $\mathsf{TOTP}_{\mathrm{home}}$, binding the execution of the protocol to the home.

The secret sk only exists on the IoT devices, which are fixed in the home; outside of the home (*i.e.*, $\mathsf{loc} \neq \text{home}$), sk is unavailable, and the protocol fundamentally cannot execute, even under compelled access. Note that this setup phase is similar to the one of standard TOTP—instead of a user downloading sk into a smartphone app or token, they download it to their smart home.

Authenticating to a service with $\mathsf{TOTP}_{\textbf{home}}$. Whenever a user requests authentication, the authorized device and IoT devices execute $\mathsf{TOTP}_{\text{home}}$, as depicted in 4.3b. If the user is at home, the authorized device contacts the home network with an authentication request along with the current timestamp value t. The parties then run an MPC protocol Π to execute the PRF of TOTP, with each party providing their share of sk as input. The output O is reconstructed on the authorized device, which computes O mod 10^6 as the 6-digit OTP. The server uses its copy of sk and PRF to compute the OTP, and validate it against the user's OTP. From the user and server's perspective, this interface—computing and verifying 6-digits OTPs—is the same as traditional RFC 6238 TOTP [M'R+11]; only the underlying PRF algorithm is changed to better support SocIoTy and its at-home properties.

4.5.2 At-Home Encryption

We now proceed to building at-home encryption. Given a simple block cipher Enc:

$$\mathsf{Enc}_{\text{home}}(m, \mathsf{loc}) = \begin{cases} \mathsf{Enc}(sk, m) & \text{if } \mathsf{loc} = \text{home} \\ \perp & \text{otherwise} \end{cases}$$

and, with corresponding decryption algorithm Dec:

$$\mathsf{Dec}_{\text{home}}(m, \mathsf{loc}) = \begin{cases} \mathsf{Dec}(sk, m) & \text{if } \mathsf{loc} = \text{home} \\ \perp & \text{otherwise} \end{cases}$$

Again, sk is only available if $\mathsf{loc} = \text{home}$.

Encrypting a file with $\mathsf{Enc}_{\mathbf{home}}$. Building $\mathsf{Enc}_{\text{home}}$ from Enc requires adding location as a precondition for executing an encryption. Our workflow for $\mathsf{Enc}_{\text{home}}$ with SocIoTy is shown in Figure 4.4.

When the user requests an encryption of a message m, it communicates this request to all of the IoT devices. The IoT devices, in turn, generate random shares, which are treated as shares of an encryption key sk. The IoT devices then run an MPC protocol Π to encrypt m, returning the encrypted file c_m. This encrypted file c_m can be stored on a cloud storage server, and all files m, c_m are effaced from the user's device after execution. Decryption operates in reverse, with the authorized device downloading c_m from cloud storage, and running $\mathsf{Dec}_{\text{home}}$ to retrieve m.

Public-key encryption. Optionally, public-key encryption can allow the Enc function to be executed "in the clear" (outside any MPC) directly on the authorized device. Decryption, however, will depend on a secret-shared corresponding private key.

Our protocol successfully binds the execution of $\mathsf{Enc}_{\text{home}}$ and $\mathsf{Dec}_{\text{home}}$ to the home itself, as it requires the IoT devices that hold sk to participate. Compelled access of the authorized device outside of the home does not lead to immediate decryption of the user's data, as the data is (1) only available in encrypted form on the cloud and (2) the authorized device never sees sk.

A downside of this solution is that it requires the IoT devices to encrypt the *entirety* of m. This may be reasonably efficient for smaller m, but as the size of m gets larger (occupying several blocks), the overhead of computing the ciphertext may become burdensome and lead to unacceptably low performance. Additionally, the IoT devices must store state linear in the number of files encrypted by the home – the more files they encrypt, the more they store.

Layering Enc and $\mathsf{Enc}_{\mathbf{home}}$. To combat this problem and more effectively scale our solution to larger data, we propose to use the smart home essentially as a *key store*. Instead of having the IoT devices encrypt an entire file, the authorized device generates an $sk_{m'}$ for each file m' it wishes to encrypt. Then, it submits $sk_{m'}$ to the IoT devices for $\mathsf{Enc}_{\text{home}}$. The IoT devices use shares of a master key sk' to encrypt $sk_{m'}$, outputting an encrypted key $c_{sk_{m'}}$. The authorized device then runs Enc on m using $sk_{m'}$ to get an encrypted file $c_{m'}$. The authorized device uploads the pair $c = (c_{sk_{m'}}, c_{m'})$ to the cloud storage, effacing m', $sk_{m'}$, $c_{m'}$, $c_{sk_{m'}}$ afterwards. To decrypt, the authorized device downloads the pair c, and uses $\mathsf{Dec}_{\text{home}}$ (via the IoT devices) on $c_{sk_{m'}}$ to retrieve $sk_{m'}$. The device proceeds to decrypt $c_{m'}$ using Dec and $sk_{m'}$ to retrieve the original file m'.

Layering encryption from the smartphone and encryption from the smart home has important performance benefits. Smartphones typically have cryptographic accelerators [ZJG22] which improve encryption and decryption speed. This reduces the burden on the IoT devices, as they only have to encrypt $sk_{m'}$, which is the size of two blocks (32 bytes). We can therefore run the block cipher in CTR mode twice to do encryption to get the ciphertext. At the same time, this process maintains the location-binding property, as decryption of c_m is predicated on decrypting $c_{sk_{m'}}$, which in turn is dependent on $\mathsf{Dec}_{\text{home}}$ and the sk' located on the IoT devices. We also eliminate the requirement to store additional state on each IoT device, as they only need to hold shares of sk', rather than every sk for all m encrypted.

$\mathsf{Enc}_{\text{home}}$ for self-revocable encryption. As previously discussed, our at-home encryption construction is well-suited to self-revocable encryption. Specifically, the "revocation cache" – a store of secrets used to re-enable access to revoked files from the cloud – can be implemented using $\mathsf{Enc}_{\text{home}}$. All of the backup key material used to recover after revocation could be encrypted and stored using $\mathsf{Enc}_{\text{home}}$. At-home encryption therefore allows non-expert users to apply a high-value security primitive like self-revocation

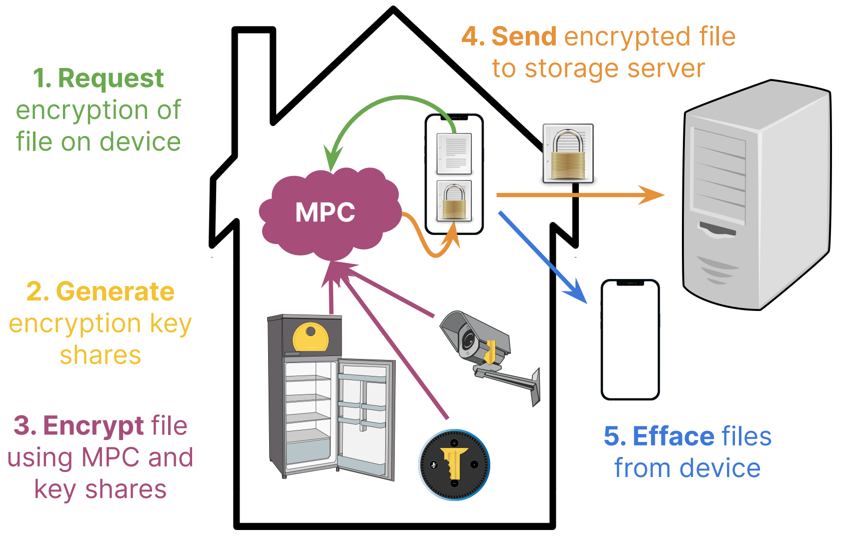

1. Request encryption of file on device
2. Generate encryption key shares
3. Encrypt file using MPC and key shares
4. Send encrypted file to storage server
5. Efface files from device
MPC

Figure 4.4: SocIoTy for at-home encryption: request, generate, encrypt, send, efface.

to their cloud storage, while re-using hardware they already have. We also gain the security property of location-binding on top of the guarantees of a system like $\mathsf{Enc}_{\text{home}}$.

4.5.3 Handling Device Churn

SocIoTy is designed to leverage existing IoT devices. While this means our solution can be easily deployed, it also means that we must adapt to changing smart home conditions. Users purchase devices for their intended use, not for at-home cryptography; a device's actual function in the smart home may conflict with their use as a member of SocIoTy. This may lead to *churn*, where an IoT device enters or leaves the SocIoTy network temporarily or permanently.

SocIoTy needs to handle two types of churn. The first type, "in-protocol churn," refers to events that cause an IoT device to stop participating in a SocIoTy operation during the middle of MPC protocol execution. The second type, "out-of-protocol churn," refers to events where a device is removed from a smart home entirely, or where a device is newly installed in the smart home. We discuss our solutions to churn below, and the security implications of our solution in §4.8.

In-protocol churn. IoT devices are designed to perform smart home tasks for their users; SocIoTy uses these same devices to provide at-home cryptography. Ideally, when running a SocIoTy computation, the IoT device is idle, and can devote its processing and network capabilities entirely to the task at hand. However, this may not be strictly the case; in the middle of computing $\mathsf{TOTP}_{\text{home}}$ or $\mathsf{Enc}_{\text{home}}$, one of the component devices may have to respond to some event triggered by the smart home. For example, the user could start interacting with the device (*e.g.*, someone opens a door or flips a switch), the IoT device might receive an important message from its associated cloud (*e.g.*, a firmware update or command), or an external event (*e.g.*, the device detects motion and starts recording).

These events may require some or all of the IoT device's resources, which are already constrained. To maintain utility in the smart home, the IoT device may be forced to stop SocIoTy computation and focus its processing capability to smart home functions instead. We refer to this as *in-protocol churn*, as a device is forced to leave in the middle of executing the MPC protocol.

A naïve way of handling this would be to pause execution of the MPC until the IoT device is idle once again. This may not be feasible, however, since the IoT device does not necessarily know how long it will be servicing the event. For instance, after a motion sensor is triggered, a camera could keep recording for several minutes. Since MPC protocols operate in lock-step, the pause in execution would stall the entire protocol until the camera is able to continue. Moreover, an IoT device cannot necessarily predict the timing of this event, which means it and the other devices cannot easily "prepare" for a pause in MPC computation.

Instead, we utilize an MPC system that allows for *dynamic participation*—entering and exiting—as a part of protocol execution. That way, a party (IoT device) that must leave can do so without seriously impacting protocol execution. As discussed in §4.2, several recent works have created models and protocols for MPC with dynamic participation [Cho+21; Gen+21]. Choudhuri et al. [Cho+21] proposed the *Fluid MPC* model, a model in which the execution of an MPC protocol is divided into discrete stages known as "epochs" such that a different (potentially overlapping) set of parties, called a "committee," participates in each epoch. The authors then design a Fluid MPC protocol—a variant of BGW [BGW88]—that achieves what they term *maximal fluidity*, where each epoch has only one round of communication.

In practical terms, this means that a party p participating in committee c can drop out of the protocol, securely transferring state to a new committee $c+1$ (not including p) without interfering with the protocol's progress or incurring notable overhead. The mechanism for state transfer, termed π_{trans} in [Cho+21], is essentially *re-sharing*. Each party in c generates a new sharing of their current shares, creating shares-of-shares. These shares-of-shares are sent to all parties in $c+1$. The parties in $c+1$ (some of which may have been in c) can then use these shares-of-shares to create regular protocol shares to continue execution. This requires communication quadratic in the size of both committees; to ensure security, a party p cannot simply send raw share data to a different party p', so we must involve *all* parties.

While the Fluid MPC model was originally designed for long-running, e.g. scientific computation, the maximal fluidity property solves our in-protocol churn problem with low overhead. Suppose an IoT device p receives a smart home event during the SocIoTy computation. The device p selects a different IoT device p', which has previously indicated that it is available for execution, and instructs all other devices that, in the next epoch, p' will be replacing p. Then, all parties participate in the π_{trans} sub-protocol, and after $O(n^2)$ communication (with small n, only the size of the home IoT network), p' will have the relevant share data to continue the execution.

Because the protocol proposed in [Cho+21] achieves maximal fluidity, hand-offs incur no additional rounds of communication. The new set of executing devices (including p') can continue computing, while p is free to handle some external event, all with minimal overhead. Also, this extra communication is only required when a device wishes to exit the network; otherwise, the protocol's communication cost is the same as regular BGW. We utilize this technique to handle in-protocol churn, and evaluate its performance in §4.6.3.

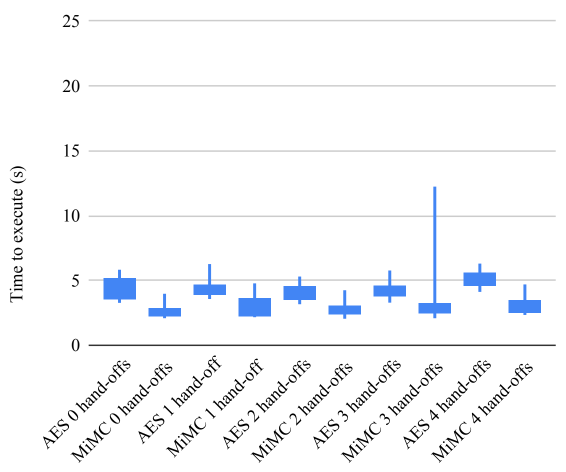
Out-of-protocol churn. While SocIoTy assumes that devices are stationary in the home, over time, *what* devices are in the home can change. Users purchase IoT devices to add new functionality to their smart home. On the other hand, IoT devices can be removed from the home, and even potentially sold. Often, both occur: an old IoT device, potentially with user data, is replaced by a new model, yet to be configured. We wish to maintain functionality of SocIoTy in the face of this *out-of-protocol churn*. This raises similar issues as topics in device provisioning and de-provisioning explored in the IoT literature [Kha+19; GG22].

First, we consider the device addition case. When the user adds a new device to their smart home, they may want to configure the device as a SocIoTy node. The new device has to gain access to long term state (i.e. key shares) in order to participate in future MPC protocol executions. More formally, if there are N devices in the original smart home the shares are parameterized by (t, N), for a threshold t. So, we must convert the (t, N) shares into $(t, N+1)$ shares to include the new device. We can do this by re-sharing as described above, with each original device computing a new share for every other device, including the new one.

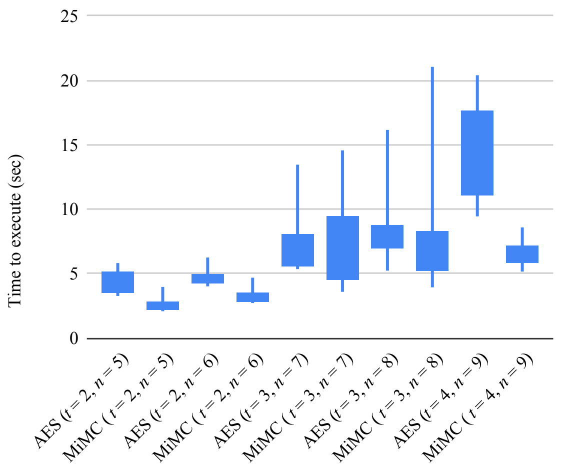
We also want to consider the device removal case, as the shares that a device holds may be necessary for the rest of the network to continue. In this instance, we can also apply re-sharing, since we are converting the (t,N) shares into $(t,N-1)$ shares. If we are replacing an old device with a new one, we can perform these operations in sequence: the old device re-shares its secrets to the new set of smart home device and the new device receives its shares as a part of the new smart home setup.

In a sense, this operation is a committee change in the Fluid MPC sense; the major difference is that the change happens *outside* of an actual protocol execution. Unlike in-protocol churn, we cannot immediately assume that another device is ready to take over for the leaving device. This means that care must be taken to potentially adjust t as devices come in and out. We discuss this further in §4.8.

Non-cooperative churn. The descriptions of churn above handle cases where churn is *cooperative*, *i.e.*, a device gracefully leaves the system and hands off state to the other parties, either in or out of the protocol. However, IoT networks can undergo *non-cooperative* churn, in which devices are forcefully removed from the network. Perhaps a device undergoes a power failure during the protocol execution, or a device is unplugged and thrown away without re-sharing anything. If non-cooperative churn occurs, and there are still enough devices online to choose n such that less than or equal to t are malicious, the protocol can continue but it must restart. However, if it is impossible to get this property, then the underlying MPC cannot continue, and the system must stall until more parties are online. We expect that smart home networks, for the most part, will exhibit cooperative churn—devices leaving to service user requests, or devices formally added or removed—far more often than non-cooperative churn. SocIoTy would allow for a protocol to continue when it encounters this more common type of churn. In this way, SocIoTy represents an improvement over the status quo, which does not



(a) Time distributions for varying number of parties exiting, with $(t = 2, n = 5)$.



(b) Time distributions for varying the (threshold, party size) pairs.

Figure 4.5: Evaluation results on 50 runs of $\mathsf{TOTP}_{\text{home}}$ for in-protocol and out-of-protocol churn on AES256 and MiMC.

support any type of churn at all.

4.5.4 Security Intuition

We now provide an intuitive overview of how our cryptosystem satisfies the property of location-binding. A more detailed analysis can be found in §4.8.

Recall that there are three main attacker scenarios we are concerned about: a remote attacker that compromises the authorized device, an unauthorized attacker *in the home* that cannot compromise the authorized device, and a remote attacker compromising the authorized device and some subset of devices in SocIoTy. A remote attacker that only compromises the authorized device is incapable of communicating with devices deployed with our protocol. Because the device does not maintain long-term secrets aside from one to authenticate to the network, the attacker gains no advantage in violating the security of the location-bound cryptography.

Because we use point-to-point secure channels with client authentication, a passive (monitoring) attacker on the network cannot learn anything about the input or output of the MPC protocol. Further, by assumption adversaries can only compromise up to t devices out of N, where the threshold t is equivalent to the threshold in Π. Thus, they cannot recover sufficient secret shares to compute the encryption or authentication keys. Nor can they force an incorrect output of the protocol if it is malicious-secure. A remote attacker who gains access to both the authorized device and at least one IoT device may attempt to impersonate the authorized user to the network. However, initiating the protocol requires low-range authentication to an offline device, which a remote attacker is incapable of interacting with. In each pertinent threat model, location binding is preserved through the confidentiality of the MPC protocol, cryptographic authentication, or physical proximity.

4.6 Evaluation

In this section we discuss our implementation of SocIoTy and our evaluation which uses it to validate the feasibility and resilience properties of our design.

4.6.1 Implementation

Our SocIoTy implementation is divided into two components. In the first, we generate circuits (MPC encodings of functions) for each at-home crytographic primitive. Then, we provide these circuits to the second component to execute them.

We define our MPC circuits in a Python-based domain-specific language we developed. The output of this step is the circuit for a given file, which is a JSON object containing the necessary information about the circuit (*i.e.*, inputs, outputs, and gates), along with *layers*, which track dependencies between gates, and *references*, which track how many times a gate is used in the circuit. This extra information helps reduce both computation and communication in SocIoTy to occur only when necessary. An example DSL circuit definition can be found in Listing 1, with its associated circuit in Listing 2.

Our circuit evaluator loads such a circuit for execution. It is written in Rust, chosen for its memory-safety, strong support for asynchronous programming, and (via LLVM) support for a common IoT architectures, including ARM and MIPS. The implementation consists of a BGW MPC [BGW88] implementation with extensions to support churn derived from Fluid MPC [Cho+21]. The underlying MPC requires communication between nodes to compute the result of a multiplication (referred to as π_{mult} in [Cho+21]), as well as to hand-off shares to another party if exiting the network (π_{trans} in [Cho+21]). An external function or library can invoke this hand-off asynchronously, making our implementation suitable for use as a part of a larger executable. Communication between device is handled via a gRPC [WZZ93] protocol that transmits share data between parties. Evaluation uses layer and reference data from the circuit to free data that is no longer needed in for the MPC computation; this helps resource-constrained IoT devices conserve both memory and bandwidth.

Our implementation is an end-to-end realization of SocIoTy that is both modular and extensible. As MPC research continue to evolve, we can generate new, more efficient circuits; as faster numerical libraries are available, we can modify our evaluator to take advantage of them. Our implementation may be of independent interest for researchers investigating new directions in at-home cryptography, or for those building other applications of MPC that involve churn.

```
x = inputs[0][0] # party 0, input share 0
y = inputs[1][0] # party 1, input share 0
output([(x * x) * (y + 4)]) # circuit output
```

Listing 1: DSL sample for a simple circuit.

```
{"field": 8, "wires": 5, "refs": {"0": 2},
"parties": [0, 1], "inputs": {"0": [0], "1": [1]}, "outputs": [4],
"constants": ["4"], "gates": [{"op": "const add", "inputs": [1, 0],
"output": 3}, {"op": "mul", "inputs": [0, 0], "output": 2},
{"op": "mul", "inputs": [2, 3], "output": 4}], "layers": [0, 2]}
```

Listing 2: The associated circuit output for Listing 1.

4.6.2 Experimental Setup

With our evaluation, we wish to answer three primary questions. First, how practical are our instantiations of at-home cryptography primitives? Next, how does our solution scale with the number of participants in the network (and out-of-protocol churn)? Finally, how does out solution scale with devices leaving and hand-off to other devices (in-protocol churn)? To this end, we set up our experiments under a simulated smart home. We configured nine Raspberry Pis, all connected via wireless LAN and running Raspberry Pi OS 11 Lite. Six units were Raspberry Pi 3B+ devices, with 64-bit ARM Cortex-A53 processors running at 1.4 GHz and 1 GB of RAM. The remaining three were Raspberry Pi 2B devices, with 32-bit ARM Cortex-A7 processors running at 900 MHz and 1 GB of RAM. The difference in capabilities represents the type of heterogeneous hardware environments common in smart homes. Additionally, these units have been used in prior works (*e.g.*, [Kum+19]) to model smart home devices.

To evaluate both at-home authentication and at-home encryption, we developed three appropriate circuits: $\mathsf{TOTP}^{\mathsf{AES256}}_{\text{home}}$, $\mathsf{TOTP}^{\mathsf{MiMC}}_{\text{home}}$, and $\mathsf{Enc}_{\text{home}}$. Each of these circuits are loaded into our evaluator, and executed on the devices.

One execution of $\mathsf{TOTP}_{\text{home}}$ involves evaluating either $\mathsf{AES256}$ or MiMC on one input timestamp ts, outputting the first 3 bytes (24 bits) for use as an OTP. Note that one execution of $\mathsf{Enc}_{\text{home}}$ is actually (almost) identical to $\mathsf{TOTP}^{\mathsf{AES256}}_{\text{home}}$, as it constitutes two $\mathsf{AES256}$ circuit executions, generating two blocks of $\mathsf{AES256}$. As such, we focus on $\mathsf{TOTP}_{\text{home}}$ in our evaluation; $\mathsf{Enc}_{\text{home}}$ can simply be extrapolated by doubling the execution time of $\mathsf{TOTP}_{\text{home}}$.

4.6.3 Results

Charts representing our results can be found in Figure 4.5. Associated summary statistics can be found in §4.7.

Base primitives. To gauge the baseline performance of our system, we tested $\mathsf{TOTP}_{\text{home}}$ in a $(t = 2, n = 5)$ configuration. We discovered that, over 30 runs, $\mathsf{TOTP}_{\text{home}}^{\mathsf{AES256}}$ took $\bar{x} = 4.37$ seconds $(s = 0.77)$ to complete execution, where $\bar{x}$ denotes the mean and s denotes the standard deviation. $\mathsf{TOTP}_{\text{home}}^{\mathsf{MiMC}}$, on the other hand, took $\bar{x} = 2.63$ seconds $(s = 0.47)$ to complete over 30 runs.

In-protocol churn. We want to assess how a node handing off to another node impacts the execution time of the entire circuit. For these tests, we have $i \in [1,2,3,4]$ nodes exit the protocol after the first epoch, trigger a state hand-off. This covers the case where execution begins, but a node realizes that it needs to spend time handling a smart home event immediately after. We ran these tests on each instantiation of $\mathsf{TOTP}_{\text{home}}$, and provide our results in Figure 4.5a. Note that the increase in standard deviations for $n > 6$ is likely due to the use of Raspberry Pi 2 devices for parties 7–9, which are lower performance than the Pi 3 devices used for parties 1–6. We also ran a test to determine if the point at which a party hands off matters (*e.g.*, if handing off 100 layers into the protocol was slower than 5 layers in), but we found that there were no noticeable differences.

Out-of-protocol churn. To help us understand the impact of out-of-protocol churn on the execution time of our solution, we also want to test different party sizes. We set

$t = \lfloor \frac{n-1}{2} \rfloor$ to maintain the highest threshold possible given the assumptions set forth in our threat model (§4.3). Figure 4.5b shows the execution time distributions for this test, with our original $\mathsf{TOTP}_{\text{home}}$ execution on $(t = 2, n = 5)$ as a baseline. As the party size increases, the median of each distribution appears to increase quadratically. This follows from the underlying BGW protocol, which has communication complexity $O(n^2)$. It also appears that, as above, the standard deviations for $n > 6$ is larger, likely for the same reason as above (slower Pi 2 devices).

4.7 Summary Statistics

Summary statistics for the evaluation distributions can be found in Tables 4.1 and 4.2.

Table 4.1: Summary statistics for the distributions in Figure 4.5a.

Hand-offs	$\mathsf{TOTP}^{\mathsf{AES256}}_{\text{home}}$	$\mathsf{TOTP}^{\mathsf{MiMC}}_{\text{home}}$
0	$\bar{x} = 4.37$, $s = 0.77$	$\bar{x} = 2.63$, $s = 0.47$
1	$\bar{x} = 4.35$, $s = 0.57$	$\bar{x} = 2.79$, $s = 0.67$
2	$\bar{x} = 4.04$, $s = 0.53$	$\bar{x} = 2.77$, $s = 0.52$
3	$\bar{x} = 4.25$, $s = 0.56$	$\bar{x} = 3.31$, $s = 1.88$
4	$\bar{x} = 5.06$, $s = 0.63$	$\bar{x} = 3.07$, $s = 0.59$

Table 4.2: Summary statistics for the distributions in Figure 4.5b.

Parties	$\mathsf{TOTP}^{\mathsf{AES256}}_{\text{home}}$	$\mathsf{TOTP}^{\mathsf{MiMC}}_{\text{home}}$
$(t = 2, n = 5)$	$\bar{x} = 4.37$, $s = 0.77$	$\bar{x} = 2.63$, $s = 0.47$
$(t = 2, n = 6)$	$\bar{x} = 4.70$, $s = 0.53$	$\bar{x} = 3.25$, $s = 0.50$
$(t = 3, n = 7)$	$\bar{x} = 7.31$, $s = 2.10$	$\bar{x} = 6.74$, $s = 3.11$
$(t = 3, n = 8)$	$\bar{x} = 8.30$, $s = 2.18$	$\bar{x} = 7.64$, $s = 3.48$
$(t = 4, n = 9)$	$\bar{x} = 14.09$, $s = 3.63$	$\bar{x} = 6.59$, $s = 0.93$

4.7.1 Discussion.

Our experimental results show that our prototype SocIoTy implementation is practical for our use cases. $\mathsf{TOTP}^{\mathsf{MiMC}}_{\text{home}}$ clearly outperforms $\mathsf{TOTP}^{\mathsf{AES256}}_{\text{home}}$, which makes sense, as

MiMC is a cipher designed with MPC applications in mind, while $\mathsf{AES256}$ was designed to be more general-purpose. While MiMC is likely the appropriate choice for instantiating SocIoTy, AES is also performant inside of the MPC, making it potentially suitable for legacy applications.

In-protocol churn is handled without a large delay on the execution of the protocol, both in terms of number of parties handing off, and point at which handoff is conducted. As for out-of-protocol churn, as the party size increases from $n = 5$ to $n = 9$, we see an increase from 4.37 seconds to 14.09 for AES, and 2.63 seconds to 6.59 for MiMC. RFC 6238 [M'R+11] recommends a time interval of 30 seconds per OTP and uses a sliding window to decide which OTPs are acceptable. The TOTP RFC recommends that servers allow OTPs from the previous two time intervals. As such, our results for all party counts all fall within a single OTP interval. A user can thus request an OTP, have SocIoTy generate it, and input it, all within the standard acceptability window.

SocIoTy is envisioned to support high-value online services and data. So. while we aim for the best performance possible, the user should be able to tolerate waiting for their most security sensitive tasks; in the worst case, a user could also attempt to submit the *next* timestamp ts' (instead of the current ts) to $\mathsf{TOTP}_{\text{home}}$; by the time $\mathsf{TOTP}_{\text{home}}$ evaluates, the OTP for ts' will be valid. Future work that develops more efficient maximally-fluid, dynamic MPC protocols or more efficient PRFs can be applied to SocIoTy, netting us instant performance benefits.

These results can also be informative for a SocIoTy deployment. A device can use the hand-off period length to decide if it wants to commit to a hand-off. While a device cannot predict in general if servicing an event will stall a SocIoTy execution, it can use estimates in its decision making. A user may choose a particular threshold and party size to minimize their delay to an acceptable level while maintaining the highest possible federation of trust.

4.8 In-Depth Security Analysis

In this section we briefly recount our threat model and then discuss how SocIoTy accounts for the various assumptions and security goals we have. Recall that we would like to ensure *location-binding*, meaning that our system should be able to provide at-home encryption and authentication even when a remote adversary compromises some minority of devices or a physically close unauthorized adversary attempts to subvert the system (see 4.3).

Consider an adversary that wishes to violate the location-binding property. Recall that they can only compromise t of the N IoT devices in the smart home which means at most t of the n parties participating in the MPC protocol Π can be compromised. By the security of the MPC protocol, if Π terminates and produces output, the output is a correct evaluation of the function over the inputs supplied and the input key shares remain private. While a corrupted party might supply incorrect input shares to the MPC, these incorrect values would simply cause reconstruction to fail. Intuitively, this is because the input to the protocol is a valid secret sharing with threshold parameter t and the "correct" input is fixed in the views of the honest parties. Thus when the protocol Π is compiled to be malicious-secure, we prevent corrupted parties from forcing the output to be incorrect, whether by deviating from the protocol or by supplying incorrect input shares.

We securely realize point-to-point channels using TLS with client-side authentication. This prevents both a remote and local attacker from seeing messages between other parties that could lead to key compromise. TLS also provides protection against replay attacks. Both of these properties prevent a local attacker on the network from violating authorization since they can neither naively see MPC output nor impersonate people on the network. Protecting against an attacker on the home network itself but who does not have access to the authorized device is somewhat easier. Because secure point-to-point channels exist between all parties, the authorized smartphone may authenticate itself to all devices in the smart home. This makes it very difficult for an unauthorized device to impersonate any device in the network, and thus privacy is maintained. Note that this covers both malicious mobile devices that might be on the network (*e.g.*, a malicious house guest) and new IoT devices maliciously installed in the home.

Having discussed IoT device compromise, we now analyze the impact of compromising an authorized device outside the home. Note that an authorized device outside the home that has not compromised a member of SocIoTy cannot communicate over the LAN, preventing it from running the previously described protocols. If an IoT device is also compromised, the attacker will not be able to initiate the protocols because it is not authenticated over NFC. Moreover, the authorized device securely deletes all secret state upon completion of the provisioning phase and after each iteration of an at-home cryptography application, meaning that access to the IoT devices is necessary to perform at-home cryptography operations.

Recall the remote service is assumed to be honest. However, in the case of at-home encryption, we can relax this assumption to *semi-honest*, as we intend to encrypt the files stored on the server. This assumption allow us to realize at-home encryption using only an IND-CPA secure encryption scheme. If a user wishes to protect themselves against a malicious cloud, at-home encryption must be realized with a non-malleable encryption system, such as an AEAD. Without an AEAD, an adversary will likely have access to a decryption oracle that allows them to recover plaintext, a vulnerability that has been demonstrated many times in the literature (*e.g.*, [Can+03b; Mey+14;

Gar+16b; BSY18; BZG20b; LGR21; BHP22; RH22]).

Churn. The security of in-protocol churn reduces to the security of π_{trans}, and we refer to [Cho+21] for that analysis. The out-of-protocol churn case is slightly more subtle, especially in IoT device removal. We assume that the user performs a factory reset of their IoT device before selling or disposing of it, as is standard practice. However, we do not assume that IoT devices have secure deletion capabilities. A malicious IoT device purchaser could extract previously held memory, retrieving a share of the key. This would in effect be compromising that device, and so without more than t shares, this adversary would not be able to reconstruct any protocol secrets. Moreover, SocIoTy renews its secrets whenever a new device is added to the smart home or a device is removed, so corrupting many discarded devices over time will not allow the user to slowly compromise secrets. As such, security is preserved during the device removal case.

Out-of-protocol churn also introduces potential concerns around MPC thresholds. We assume that the threshold is set at $t = \lfloor \frac{n-1}{2} \rfloor$ where $n < N$ (*i.e.*, the number of SocIoTy devices is less than the total number of devices in the smart home). As devices are added and removed, we may need to adjust n. To maintain security against the same fraction of malicious parties, t will need to be modified accordingly.

Achieving malicious security. While not considered thus far, we additionally note that Π can be instantiated to be malicious-secure, meaning that even if an adversary can completely compromise a device and force it to engage in arbitrary behavior, our protocol is still secure. The protocol described in [Cho+21] achieves malicious security by using a variant of the malicious security check proposed by Chida et al. [Chi+18], which maintains a succinct amount of state per party that does not grow with the size of the circuit nor add additional rounds to the MPC protocol. This works for MiMC, which operates in $GF(2^{128})$; Unfortunately, AES256 computes over a small field ($GF(2^8)$), so we cannot use their check directly. However, the end of Appendix B in [Cho+21] presents a solution for operating over small fields: run the security check many times in parallel to amplify security to the desired level. To achieve a security level of at least 60 bits (actual security equal to 64) we can execute that same check 8 times in parallel for the same effect as using the 61 bit Mersenne field used in [Chi+18]. The cost for doing this is a constant multiplicative overhead in the state that must be held by each party during the protocol.

Non-goals. The MPC protocol Π that we use does not provide guaranteed output delivery, a property which ensures that adversarial deviation (including aborting) cannot prevent an honest party from learning the protocol output. As such, it is possible for a remote attacker to mount a denial of service attack.

4.9 Related Work

We conclude this constructive contribution with consideration and comparison with prior works with similar goals. We summarize our comparisons in Table 4.3, with the following abbreviations: "Authn" notes if the work supports authentication, "Enc" encryption, "Loc" if location can be cryptographically enforced, "HW" if the solution requires novel/custom hardware to be used, and "Non-expert" if the solution specifically designs around non-expert user requirements to facilitate adoption and practicality.

Location-based cryptography. Heuristics around location-based cryptography were originally formed in the networking community, with a set of "geo-encryption" algorithms [SD03; Qiu+07; FV12; AFAI07] 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. [Cha+09] as "position-based

Table 4.3: A comparison of related work with similar goals to those of SocIoTy.

	Authn	Enc	Loc	HW	Non-expert
Geo-encryption [SD03; Qiu+07; FV12; AFAI07]	✗	✓	✓	✓	✗
Position-based crypto [Cha+09; Bro+17; Buh+11]	✓	✓	✓	N/A	✗
Time-specific encryption [Phu+19; PQ10; Kas+12]	✗	✓	✓	N/A	✗
HSMs [WG11; Hup+20; Han+19]	✓	✗	✓	✗	✗
Wearable devices [Cao+20; Chu14; SS18]	✓	✗	✗	✗	✓
Proximity measurement [Zha+17; ABS18]	✓	✗	✗	✗	✗
SocIoTy	✓	✓	✓	✗	✓

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. Works since have explored the assumptions made by Chandran and their implications in complexity theory [Bro+17] and in the quantum setting [Buh+11]. Further, Phuong et al. developed a location-based encryption scheme in 2019 [Phu+19], 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 [PQ10; Kas+12] to ensure decryption only at particular points for a given ciphertext.

Hardware security modules (HSMs). Hardware security modules are separate, dedicated computing devices that protect cryptographic keys by storing them and mediating their use. They provide tamper-evidence or even tamper-resistance through the use of specialized hardware [Kas18]. Once tampering is detected, the device may stop functioning appropriately or delete its secret keys. HSMs are commonly used to protect keys used by certificate authorities, banks, and cryptocurrency providers. They are present within vehicles [WG11], operational technology [Hup+20], and cloud applications [Han+19]. 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 [Kas18].

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. [Zha+17] describe an easier authentication for IoT devices by gesturing with a smartphone in close proximity to the devices. Aman et al. [ABS18] used 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, [Cao+20; Chu14; SS18] use wearable IoT devices as a second-factor for authentication.

Chapter 5

Conclusion

Throughout the course of this work, we have proposed new approaches to analyzing the security of cryptographic schemes, starting with encryption and extending to modern cryptographic tools such as MPC. The unifying goal of these efforts was not only to advance the field of attacks development and security analysis, but also to provide usability and convenience for experts and non-expert practitioners alike to simultaneously expand or democratize access to these aspects of cryptographic knowledge. We then continued this line of reasoning forward to develop a new secure protocol which attempts to address practically-motivated needs for the end-user, similarly driven by the guiding principle of accessibility which we realize by using commodity non-specialized hardware to achieve security.

Our explorations elucidated a number of open problems, some of which we were able to resolve. We briefly enumerate the remainder as a forward-looking conclusion to this work.

1. In Chapters 2 and 3 we establish the need for further theoretical treatment of the empirical advancements we make in applying approximated `Max#SAT`. Our approaches necessarily weaken some theoretical guarantees of previously known approximation algorithms, but in exchange are able to be practically feasible for

meaningful formulae. A deeper analysis of the underlying complexities and costs of our methods and heuristics would better inform extrapolations of our methods as well as identify specific areas in which the SAT solving community could focus for the mutual benefit of their work and ours.

2. In Chapter 3, having successfully assuaged the main limitation of the methods of Chapter 2, we identify a horizon of new classes of problems our methods may be applied to. We do not know the extent that this horizon could reach, however, we note that it certainly at least includes a few promising areas. First, differential privacy thresholding and security analysis; and second, evaluation and automated exploitation of side channel attacks.
3. In Chapter 4, we present an exemplary secure protocol which bridges modern cryptographic tools with end-user accessibility. Arguably this chapter's *primary purpose* is to suggest and reinforce the emerging line of research in which cryptography experts attempt to not only expand the bounds of what is theoretically possible, but also translate those advancements into systems and tools which can affect organizations' and individuals' day to day lives. We hope that the summation of this work can serve as a "proof by existence" to the possibility and value of achieving both of those aims as we continue to advance the field.

Bibliography

[DH76] Whitfield Diffie and Martin Hellman. “New directions in cryptography”. In: *IEEE Trans. Inform. Theory* 22 (1976), pp. 472–492.

[Sma16] Nigel P Smart. “The enigma machine”. In: *Cryptography Made Simple* (2016), pp. 133–161.

[BB02] Steven Michael Bellovin and Randy Bush. “Security Through Obscurity Considered Dangerous”. In: *Internet Drafts of the Internet Engineering Task Force* (2002).

[Tur14] Sean Turner. “Transport layer security”. In: *IEEE Internet Computing* 18.6 (2014), pp. 60–63.

[Yao82] Andrew C Yao. “Protocols for secure computations”. In: *SFCS ’82*. 1982, pp. 160–164.

[Yao86a] Andrew Chi-Chih Yao. “How to Generate and Exchange Secrets”. In: *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*. SFCS ’86. Washington, DC, USA: IEEE Computer Society, 1986, pp. 162–167. ISBN: 0-8186-0740-8. DOI: 10.1109/SFCS.1986.25. URL: http://dx.doi.org/10.1109/SFCS.1986.25.

[BOGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. “Completeness theorems for non-cryptographic fault-tolerant distributed computation”. In: *Proceedings of the twentieth annual ACM symposium on Theory of computing*. 1988, pp. 1–10.

[Gen09] Craig Gentry. “Fully homomorphic encryption using ideal lattices”. In: *ACM STOC ’09*. 2009, pp. 169–178.

[Nak08] Satoshi Nakamoto. “Bitcoin whitepaper”. In: *URL: https://bitcoin. org/bitcoin. pdf-(: 17.07. 2019)* 9 (2008), p. 15.

[But+13] Vitalik Buterin et al. “Ethereum white paper”. In: *GitHub repository* 1 (2013), pp. 22–23.

[Swe+09] Michael Sweikata et al. “The usability of end user cryptographic products”. In: *2009 Information Security Curriculum Development Conference*. 2009, pp. 55–59.

[WT99] Alma Whitten and J Doug Tygar. “Why Johnny Can’t Encrypt: A Usability Evaluation of PGP 5.0.” In: *USENIX security symposium*. Vol. 348. 1999, pp. 169–184.

[Ruo+15] Scott Ruoti et al. “Why Johnny still, still can’t encrypt: Evaluating the usability of a modern PGP client”. In: *arXiv preprint arXiv:1510.08555* (2015).

[Rog15] Phillip Rogaway. “The moral character of cryptographic work”. In: *Cryptology ePrint Archive* (2015).

[Bel96] Steven M. Bellovin. “Problem Areas for the IP Security Protocols”. In: *Proceedings of the 6th Conference on USENIX Security Symposium, Focusing on Applications of Cryptography*. Vol. 6. San Jose, California: USENIX Association, 1996, pp. 21–21.

[NY90] M. Naor and M. Yung. “Public-key Cryptosystems Provably Secure Against Chosen Ciphertext Attacks”. In: *Proceedings of the Twenty-second Annual ACM Symposium on Theory of Computing*. STOC ’90. Baltimore, Maryland, USA: ACM, 1990, pp. 427–437. DOI: 10.1145/100216.100273.

[BN00] Mihir Bellare and Chanathip Namprempre. “Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm”. In: *ASIACRYPT 2000*. Ed. by Tatsuaki Okamoto. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 531–545. ISBN: 978-3-540-44448-0.

[FRS17] Daniel Fremont, Markus N. Rabe, and Sanjit A. Seshia. “Maximum Model Counting”. In: *Proceedings of the 31st AAAI Conference on Artificial Intelligence (AAAI)*. 2017, pp. 3885–3892.

[Mer78] Ralph C Merkle. “Secure communications over insecure channels”. In: *Communications of the ACM* 21.4 (1978), pp. 294–299.

[Tse68] Grigorii Samuilovich Tseitin. “On the complexity of proof in prepositional calculus”. In: *Zapiski Nauchnykh Seminarov POMI* 8 (1968), pp. 234–259.

[LSP19] Leslie Lamport, Robert Shostak, and Marshall Pease. “The Byzantine generals problem”. In: *Concurrency: the works of leslie lamport*. 2019, pp. 203–226.

[BFM90] Manuel Blum, Paul Feldman, and Silvio Micali. “Proving Security Against Chosen Cyphertext Attacks”. In: *CRYPTO’88*. Ed. by Shafi Goldwasser. Vol. 403. LNCS. Springer, Heidelberg, 1990, pp. 256–268. DOI: 10.1007/0-387-34799-2_20.

[BZG20a] Gabrielle Beck, Maximilian Zinkus, and Matthew Green. “Automating the development of chosen ciphertext attacks”. In: *USENIX Security ’20*. 2020, pp. 1821–1837.

[NIS24] NIST. *NIST Cryptography*. https://www.nist.gov/cryptography. 2024.

[WS+96] David Wagner, Bruce Schneier, et al. “Analysis of the SSL 3.0 protocol”. In: *The Second USENIX Workshop on Electronic Commerce Proceedings*. Vol. 1. 1. 1996, pp. 29–40.

[Dow+15] Benjamin Dowling et al. “A cryptographic analysis of the TLS 1.3 handshake protocol candidates”. In: *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*. 2015, pp. 1197–1210.

[Smi12] Michael Smith. *What you need to know about BEAST*. Available at https://blogs.akamai.com/2012/05/what-you-need-to-know-about-beast.html. 2012.

[AP13] Nadhem J. AlFardan and Kenneth G. Paterson. “Lucky Thirteen: Breaking the TLS and DTLS Record Protocols”. In: *IEEE S&P (Oakland) ’13*. 2013, pp. 526–540. DOI: 10.1109/SP.2013.42.

[Beu+15] B. Beurdouche et al. “A Messy State of the Union: Taming the Composite State Machines of TLS”. In: *2015 IEEE Symposium on Security and Privacy*. 2015, pp. 535–552.

[Adr+15] David Adrian et al. “Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice”. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. CCS ’15. Denver, Colorado, USA: ACM, 2015, pp. 5–17. ISBN: 978-1-4503-3832-5. DOI: 10.1145/2810103.2813707.

[Avi+16] Nimrod Aviram et al. “DROWN: Breaking TLS Using SSLv2”. In: *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, 2016, pp. 689–706. ISBN: 978-1-931971-32-4.

[Pod+18] Damian Poddebniak et al. “Efail: Breaking S/MIME and OpenPGP Email Encryption using Exfiltration Channels”. In: *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, 2018, pp. 549–566. ISBN: 978-1-931971-46-1. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/poddebniak.

[W3C17] W3C. *Web Cryptography API*. https://www.w3.org/TR/WebCryptoAPI/. 2017.

[JS11] Tibor Jager and Juraj Somorovsky. “How to Break XML Encryption”. In: *ACM CCS ’2011*. ACM Press, 2011.

[Mau+15] Florian Maury et al. "Format Oracles on OpenPGP". In: *Topics in Cryptology - CT-RSA 2015, The Cryptographer's Track at the RSA Conference 2015*. San Francisco, United States, 2015, pp. 220–236. DOI: 10.1007/978-3-319-16715-2_12. URL: https://hal.archives-ouvertes.fr/hal-01154822.

[Gar+16a] Christina Garman et al. "Dancing on the Lip of the Volcano: Chosen Ciphertext Attacks on Apple iMessage". In: *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, 2016, pp. 655–672. ISBN: 978-1-931971-32-4. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/garman.

[VP17] Mathy Vanhoef and Frank Piessens. "Key Reinstallation Attacks: Forcing Nonce Reuse in WPA2". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. Dallas, Texas, USA: ACM, 2017, pp. 1313–1328. ISBN: 978-1-4503-4946-8. DOI: 10.1145/3133956.3134027. URL: http://doi.acm.org/10.1145/3133956.3134027.

[Vau02] Serge Vaudenay. "Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS". In: *EUROCRYPT '02*. Vol. 2332 of LNCS. London, UK: Springer-Verlag, 2002, pp. 534–546. ISBN: 3-540-43553-0.

[Hun10] Troy Hunt. *Fear, uncertainty and the padding oracle exploit in ASP.NET*. Available at https://www.troyhunt.com/fear-uncertainty-and-and-padding-oracle/. 2010.

[Bar+12] Romain Bardou et al. "Efficient Padding Oracle Attacks on Cryptographic Hardware". English. In: *CRYPTO '12*. Vol. 7417 of LNCS. Springer, 2012, pp. 608–625. ISBN: 978-3-642-32008-8. DOI: 10.1007/978-3-642-32009-5_36.

[MDK14] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. *This POODLE bites: Exploiting the SSLv3 Fallback*. Available at https://www.openssl.org/~bodo/ssl-poodle.pdf. 2014.

[APW09] Martin R. Albrecht, Kenneth G. Paterson, and Gaven J. Watson. "Plaintext Recovery Attacks Against SSH". In: *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*. SP '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 16–26. ISBN: 978-0-7695-3633-0. DOI: 10.1109/SP.2009.5. URL: https://doi.org/10.1109/SP.2009.5.

[DP10] Jean Paul Degabriele and Kenneth G. Paterson. "On the (in)Security of IPsec in MAC-then-encrypt Configurations". In: *Proceedings of the 17th ACM Conference on Computer and Communications Security*. CCS '10. Chicago, Illinois, USA: ACM, 2010, pp. 493–504. ISBN: 978-1-4503-0245-6. DOI: 10.1145/1866307.1866363. URL: http://doi.acm.org/10.1145/1866307.1866363.

[Can+03a] Brice Canvel et al. “Password Interception in a SSL/TLS Channel”. In: *Advances in Cryptology - CRYPTO 2003*. Ed. by Dan Boneh. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 583–599. ISBN: 978-3-540-45146-4.

[Mit05] Chris J. Mitchell. “Error oracle attacks on CBC mode: Is There a Future for CBC Mode Encryption?” In: *Information Security, 8th International Conference, ISC 2005, Singapore, September 20-23, 2005, Proceedings*. 2005, pp. 244–258.

[AP15] Martin R. Albrecht and Kenneth G. Paterson. *Lucky Microseconds: A Timing Attack on Amazon’s s2n Implementation of TLS*. Cryptology ePrint Archive, Report 2015/1129. https://eprint.iacr.org/2015/1129. 2015.

[Mer+19] Robert Merget et al. “Scalable Scanning and Automatic Classification of TLS Padding Oracle Vulnerabilities”. In: *USENIX Security 2019*. Ed. by Nadia Heninger and Patrick Traynor. USENIX Association, 2019, pp. 1029–1046.

[AF18] Gildas Avoine and Loïc Ferreira. “Attacking GlobalPlatform SCP02-compliant Smart Cards Using a Padding Oracle Attack”. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2018.2 (2018), pp. 149–170. DOI: 10.13154/tches.v2018.i2.149-170. URL: https://tches.iacr.org/index.php/TCHES/article/view/878.

[Kup+15] Dennis Kupser et al. “How to Break XML Encryption – Automatically”. In: *Proceedings of the 9th USENIX Conference on Offensive Technologies*. WOOT’15. Washington, D.C.: USENIX Association, 2015.

[Jou06] Antoine Joux. *Authentication failures in NIST version of GCM*. Available at https://csrc.nist.gov/csrc/media/projects/block-cipher-techniques/documents/bcm/comments/800-38-series-drafts/gcm/joux_comments.pdf. 2006.

[MW16] John Mattsson and Magnus Westerlund. “Authentication Key Recovery on Galois/Counter Mode GCM”. In: *Proceedings of the 8th International Conference on Progress in Cryptology — AFRICACRYPT 2016 - Volume 9646*. Berlin, Heidelberg: Springer-Verlag, 2016, pp. 127–143. ISBN: 978-3-319-31516-4. DOI: 10.1007/978-3-319-31517-1_7. URL: https://doi.org/10.1007/978-3-319-31517-1_7.

[Böc+16] Hanno Böck et al. “Nonce-Disrespecting Adversaries: Practical Forgery Attacks on GCM in TLS”. In: *10th USENIX Workshop on Offensive Technologies (WOOT 16)*. Austin, TX: USENIX Association, 2016. URL: https://www.usenix.org/conference/woot16/workshop-program/presentation/bock.

[Mül+19] Jens Müller et al. *PDF Insecurity*. Available at https://www.pdf-insecurity.org/encryption/encryption.html. 2019.

[BRB18] L. Bang, N. Rosner, and T. Bultan. "Online Synthesis of Adaptive Side-Channel Attacks Based On Noisy Observations". In: *2018 IEEE European Symposium on Security and Privacy (EuroS P)*. 2018, pp. 307–322. DOI: 10.1109/EuroSP.2018.00029.

[Pha+17] Quoc-Sang Phan et al. "Synthesis of Adaptive Side-Channel Attacks". In: *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*. Santa Barbara, CA, USA: IEEE, 2017. DOI: 10.1109/CSF.2017.8.

[CSP16] Pasquale Malacaria Corina S. Pasareanu Quoc-Sang Phan. "Multi-run side-channel analysis using Symbolic Execution and Max-SMT". In: *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*. Lisbon, Portugal: IEEE, 2016. DOI: 10.1109/CSF.2016.34.

[Gan18] Vijay Ganesh. *The Simple Theorem Prover (STP)*. Available at https://stp.github.io/. 2018.

[MB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *TACAS '08*. Vol. 4963. LNCS. 2008, pp. 337–340.

[SNC09] Mate Soos, Karsten Nohl, and Claude Castelluccia. "Extending SAT Solvers to Cryptographic Problems". In: *International Conference on Theory and Applications of Satisfiability Testing (SAT 2009)*. 2009, pp. 244–257. DOI: 10.1007/978-3-642-02777-2_24.

[Nie+18] Aina Niemetz et al. "Btor2 , BtorMC and Boolector 3.0". In: *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*. Ed. by Hana Chockler and Georg Weissenbacher. Vol. 10981. Lecture Notes in Computer Science. 2018, pp. 587–595. DOI: 10.1007/978-3-319-96145-3_32. URL: https://doi.org/10.1007/978-3-319-96145-3_32.

[Val79b] Leslie G. Valiant. "The Complexity of Enumeration and Reliability Problems". In: *SIAM J. Comput.* 8(3) (1979), pp. 410–421.

[Tod91] Seinosuke Toda. "PP is As Hard As the Polynomial-time Hierarchy". In: *SIAM J. Comput.* 20.5 (1991), pp. 865–877. ISSN: 0097-5397. DOI: 10.1137/0220053. URL: http://dx.doi.org/10.1137/0220053.

[BL99] Elazar Birnbaum and Eliezer L. Lozinskii. "The Good Old Davis-Putnam Procedure Helps Counting Models". In: *J. Artif. Int. Res.* 10.1 (1999), pp. 457–477. ISSN: 1076-9757. URL: http://dl.acm.org/citation.cfm?id=1622859.1622875.

[BJP00] Roberto J. Bayardo, Jr., and J. D. Pehoushek. “Counting Models using Connected Components”. In: *In AAAI*. 2000, pp. 157–162.

[GSS06] Carla P. Gomes, Ashish Sabharwal, and Bart Selman. “Model Counting: A New Strategy for Obtaining Good Bounds”. In: *Proceedings of the 21st National Conference on Artificial Intelligence - Volume 1*. AAAI’06. Boston, Massachusetts: AAAI Press, 2006, pp. 54–61. ISBN: 978-1-57735-281-5. URL: http://dl.acm.org/citation.cfm?id=1597538.1597548.

[CMV16a] Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. “Algorithmic Improvements in Approximate Counting for Probabilistic Inference: From Linear to Logarithmic SAT Calls”. In: *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*. 2016. URL: https://bitbucket.org/kuldeepmeel/approxmc.

[SM19] Mate Soos and Kuldeep S. Meel. “BIRD: Engineering an Efficient CNF-XOR SAT Solver and its Applications to Approximate Model Counting”. In: *Proceedings of AAAI Conference on Artificial Intelligence (AAAI)*. 2019. URL: https://github.com/meelgroup/approxmc.

[VV86] L.G. Valiant and V.V. Vazirani. “NP is as easy as detecting unique solutions”. In: *Theoretical Computer Science* 47 (1986), pp. 85 –93. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(86)90135-0.

[Sto83] Larry Stockmeyer. “The complexity of approximate counting”. In: *Proceedings of the fifteenth annual ACM symposium on Theory of computing*. ACM. 1983, pp. 118–126.

[DP60] Martin Davis and Hilary Putnam. “A Computing Procedure for Quantification Theory”. In: *J. ACM* 7.3 (1960), pp. 201–215. ISSN: 0004-5411. DOI: 10.1145/321033.321034. URL: http://doi.acm.org/10.1145/321033.321034.

[DLL62a] Martin Davis, George Logemann, and Donald Loveland. “A Machine Program for Theorem-proving”. In: *Commun. ACM* 5.7 (1962), pp. 394–397. ISSN: 0001-0782. DOI: 10.1145/368273.368557.

[HS04] Holger Hoos and Thomas Sttzle. *Stochastic Local Search: Foundations & Applications*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004. ISBN: 1558608729.

[CDM17] Dmitry Chistikov, Rayna Dimitrova, and Rupak Majumdar. “Approximate counting in SMT and value estimation for probabilistic programs”. In: *Acta Informatica* 54.8 (2017), pp. 729–764. ISSN: 1432-0525. DOI: 10.1007/s00236-017-0297-2. URL: https://doi.org/10.1007/s00236-017-0297-2.

[San+04] Tian Sang et al. “Combining Component Caching and Clause Learning for Effective Model Counting”. In: *SAT 2004*. 2004.

[WS05] Wei Wei and Bart Selman. “A New Approach to Model Counting”. In: *Theory and Applications of Satisfiability Testing*. Ed. by Fahiem Bacchus and Toby Walsh. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 324–339. ISBN: 978-3-540-31679-4.

[BDP03] Fahiem Bacchus, Shannon Dalmao, and Toniann Pitassi. “DPLL with Caching: A new algorithm for #SAT and Bayesian Inference”. In: *Electronic Colloquium on Computational Complexity (ECCC)* 10 (2003).

[Cha+15] Supratik Chakraborty et al. “From weighted to unweighted model counting”. In: *Twenty-Fourth International Joint Conference on Artificial Intelligence*. 2015.

[Cha+14a] Supratik Chakraborty et al. “Distribution-aware sampling and weighted model counting for SAT”. In: *Twenty-Eighth AAAI Conference on Artificial Intelligence*. 2014.

[BFT16] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The Satisfiability Modulo Theories Library (SMT-LIB)*. `www.SMT-LIB.org`. 2016.

[RP] Glenn Randers-Pehrson. *PNG (Portable Network Graphics) Specification, Version 1.2*. URL: http://www.libpng.org/pub/png/spec/1.2/PNG-Structure.html.

[Pfe03] S. Pfeiffer. *The Ogg Encapsulation Format Version 0*. RFC 3533. RFC Editor, 2003.

[Kal98] Burt Kaliski. *PKCS #7: Cryptographic Message Syntax Version 1.5*. RFC 2315. 1998. DOI: 10.17487/RFC2315. URL: https://rfc-editor.org/rfc/rfc2315.txt.

[Zha+16] Shengjia Zhao et al. “Closing the Gap Between Short and Long XORs for Model Counting”. In: *AAAI 2016*. 2016. URL: https://arxiv.org/abs/1512.08863.

[Erm+14] Stefano Ermon et al. *Low-Density Parity Constraints for Hashing-Based Discrete Integration*. 2014.

[Moo+16] Benjamin Mood et al. “Frigate: A validated, extensible, and efficient compiler and interpreter for secure computation”. In: *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2016, pp. 112–127.

[Fra+14] Martin Franz et al. “CBMC-GC: an ANSI C compiler for secure two-party computations”. In: *International Conference on Compiler Construction*. Springer. 2014, pp. 244–249.

[Zal] Michal Zalewski. *Technical "whitepaper" for afl-fuzz*. URL: http://lcamtuf.coredump.cx/afl/technical_details.txt.

[She+18] Shiqi Shen et al. *Neuro-Symbolic Execution: The Feasibility of an Inductive Approach to Symbolic Execution*. 2018. eprint: 1807.00575.

[Wan+19] Shuai Wang et al. "Identifying Cache-Based Side Channels through Secret-Augmented Abstract Interpretation". In: *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, 2019, pp. 657–674. ISBN: 978-1-939133-06-9. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/wang-shuai.

[TB09] Erik Tews and Martin Beck. "Practical attacks against WEP and WPA". In: *Proceedings of the second ACM conference on Wireless network security*. ACM. 2009, pp. 79–86.

[Lim18] Arm Limited. *ARM Architecture Reference Manual*. 2018. URL: https://documentation-service.arm.com/static/606dc36485368c4c2b1bf62f%3F.

[Ecl20] Eclypsium. *Perilous Peripherals: The Hidden Dangers Inside Windows & Linux Computers*. 2020. URL: https://eclypsium.com/2020/2/18/unsigned-peripheral-firmware/.

[MTB20] Brendan Moran, Hannes Tschofenig, and Henk Birkholz. *An Information Model for Firmware Updates in IoT Devices*. Internet-Draft draft-ietf-suit-information-model-05. http://www.ietf.org/internet-drafts/draft-ietf-suit-information-model-05.txt. IETF Secretariat, 2020. URL: http://www.ietf.org/internet-drafts/draft-ietf-suit-information-model-05.txt.

[Sch15] Steve Schmidt. *Introducing s2n, a New Open Source TLS Implementation*. https://aws.amazon.com/blogs/security/introducing-s2n-a-new-open-source-tls-implementation/. 2015.

[Fer05] Niels Ferguson. *Authentication Weaknesses in GCM*. 2005. URL: https://csrc.nist.gov/csrc/media/projects/block-cipher-techniques/documents/bcm/comments/cwc-gcm/ferguson2.pdf.

[Rie+17] Heinz Riener et al. "metaSMT: focus on your application and not on solver integration". In: *International Journal on Software Tools for Technology Transfer* 19.5 (2017), pp. 605–621.

[Ran18] Venkatesh-Prasad Ranganath. *SAT Encoding: Solving Simpler Sudoku*. Available at https://medium.com/@rvprasad/sat-encoding-solving-simpler-sudoku-d92671206d1e. 2018.

[ppm18] ppmx. *Z3 Sudoku Solver*. Available at https://github.com/ppmx/sudoku-solver. 2018.

[Rog02] Phillip Rogaway. "Authenticated Encryption with Associated Data". In: *CCS '02*. ACM Press, 2002.

[RS06] Phillip Rogaway and Thomas Shrimpton. “A Provable-Security Treatment of the Key-Wrap Problem”. In: *EUROCRYPT 2006*. Ed. by Serge Vaudenay. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 373–390. ISBN: 978-3-540-34547-3.

[BKN04] Mihir Bellare, Tadayoshi Kohno, and Chanathip Namprempre. “Breaking and Provably Repairing the SSH Authenticated Encryption Scheme: A Case Study of the Encode-then-Encrypt-and-MAC Paradigm”. In: *ACM Trans. Inf. Syst. Secur.* 7.2 (2004), pp. 206–241. ISSN: 1094-9224. DOI: 10.1145/996943.996945. URL: http://doi.acm.org/10.1145/996943.996945.

[RD10] Juliano Rizzo and Thai Duong. “Practical Padding Oracle Attacks”. In: *Proceedings of the 4th USENIX Conference on Offensive Technologies*. 2010, pp. 1–8.

[YPM05] Arnold K. L. Yau, Kenneth G. Paterson, and Chris J. Mitchell. “Padding Oracle Attacks on CBC-Mode Encryption with Secret and Random IVs”. In: *Fast Software Encryption: 12th International Workshop, FSE 2005, Paris, France, February 21-23, 2005, Revised Selected Papers*. 2005, pp. 299–319.

[PY04] Kenneth G. Paterson and Arnold K. L. Yau. “Padding Oracle Attacks on the ISO CBC Mode Encryption Standard”. In: *Topics in Cryptology - CT-RSA 2004, The Cryptographers’ Track at the RSA Conference 2004, San Francisco, CA, USA, February 23-27, 2004, Proceedings*. 2004.

[COQ09] Nicolas T. Courtois, Sean O’Neil, and Jean-Jacques Quisquater. “Practical Algebraic Attacks on the Hitag2 Stream Cipher”. In: *Information Security*. Ed. by Pierangela Samarati et al. Berlin, Heidelberg, 2009, pp. 167–176. ISBN: 978-3-642-04474-8.

[MZ06] Ilya Mironov and Lintao Zhang. “Applications of SAT Solvers to Cryptanalysis of Hash Functions”. In: *International Conference on Theory and Applications of Satisfiability Testing (SAT 06)*. Vol. 4121. Lecture Notes in Computer Science. Springer, 2006, pp. 102–115. ISBN: 3-540-37206-7. URL: https://www.microsoft.com/en-us/research/publication/applications-of-sat-solvers-to-cryptanalysis-of-hash-functions/.

[AKMY10] Abdel Alim Kamal and Amr M. Youssef. “Applications of SAT Solvers to AES key Recovery from Decayed Key Schedule Images.” In: *IACR Cryptology ePrint Archive* 2010 (2010), p. 324. DOI: 10.1109/SECURWARE.2010.42.

[Ivr+15] Alexander Ivrii et al. “On computing minimal independent support and its applications to sampling and counting”. In: *Constraints* 21 (2015). DOI: 10.1007/s10601-015-9204-z.

[Gru+18] Paul Grubbs et al. “Pump Up the Volume: Practical Database Reconstruction from Volume Leakage on Range Queries”. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’18. Toronto, Canada: ACM, 2018, pp. 315–331. ISBN: 978-1-4503-5693-0. DOI: 10.1145/3243734.3243864. URL: http://doi.acm.org/10.1145/3243734.3243864.

[ZCG23] Maximilian Zinkus, Yinzhi Cao, and Matthew D Green. “McFIL: Model Counting Functionality-Inherent Leakage”. In: *32nd USENIX Security Symposium (USENIX Security 23)*. 2023, pp. 7001–7018.

[GMR85] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. “The Knowledge Complexity of Interactive Proof-Systems (Extended Abstract)”. In: *17th ACM STOC*. ACM Press, 1985, pp. 291–304. DOI: 10.1145/22145.22178.

[Ish+07] Yuval Ishai et al. “Zero-knowledge from secure multiparty computation”. In: *ACM STOC ’07*. 2007, pp. 21–30.

[MF06] Payman Mohassel and Matthew Franklin. “Efficiency tradeoffs for malicious two-party computation”. In: *International Workshop on Public Key Cryptography*. 2006, pp. 458–473.

[HKE12] Y. Huang, J. Katz, and D. Evans. “Quid-Pro-Quo-tocols: Strengthening Semi-honest Protocols with Dual Execution”. In: *2012 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 2012, pp. 272–284. DOI: 10.1109/SP.2012.43.

[CMS09] Michael R Clarkson, Andrew C Myers, and Fred B Schneider. “Quantifying information flow with beliefs”. In: *Journal of Computer Security* 17.5 (2009), pp. 655–701.

[Mar+11] Piotr Mardziel et al. “Dynamic enforcement of knowledge-based security policies”. In: *IEEE CSF ’11*. 2011, pp. 114–128.

[Mar+13a] Piotr Mardziel et al. “Dynamic enforcement of knowledge-based security policies using probabilistic abstract interpretation”. In: *Journal of Computer Security* 21.4 (2013), pp. 463–532.

[Mar+13b] Piotr Mardziel et al. “Knowledge inference for optimizing and enforcing secure computations”. In: *Proceedings of the Annual Meeting of the US/UK International Technology Alliance*. 2013.

[CHM04] David Clark, Sebastian Hunt, and Pasquale Malacaria. “Quantified interference: Information theory and information flow”. In: *Workshop on Issues in the Theory of Security (WITS’04)*. 2004.

[KMM13] Vladimir Klebanov, Norbert Manthey, and Christian Muise. “SAT-based analysis and quantification of information flow in programs”. In: *International Conference on Quantitative Evaluation of Systems*. 2013, pp. 177–192.

[Boy+12] Elette Boyle et al. “Multiparty computation secure against continual memory leakage”. In: *ACM STOC ’12*. 2012, pp. 1235–1254.

[HSV20] Carmit Hazay, Abhi Shelat, and Muthuramakrishnan Venkitasubramaniam. “Going Beyond Dual Execution: MPC for Functions with Efficient Verification”. In: *IACR PKC ’20*. 2020, pp. 328–356.

[Alm+21] Ghada Almashaqbeh et al. “Gage MPC: Bypassing Residual Function Leakage for Non-Interactive MPC”. In: *Proceedings on Privacy Enhancing Technologies* 4 (2021), pp. 528–548.

[Zin23] Maximilian A. Zinkus. *McFIL-Release on GitHub*. https://github.com/maxzinkus/McFIL-Release. 2023.

[HS00] Holger H Hoos and Thomas Stützle. “SATLIB: An online resource for research on SAT”. In: *Sat* 2000 (2000), pp. 283–292.

[Woo+14] Gavin Wood et al. “Ethereum: A secure decentralised generalised transaction ledger”. In: *Ethereum project yellow paper* 151.2014 (2014), pp. 1–32.

[Coo71] Stephen A. Cook. “The Complexity of theorem proving procedures”. In: *ACM STOC ’71*. 1971, pp. 151–158.

[DLL62b] Martin Davis, George Logemann, and Donald Loveland. “A machine program for theorem-proving”. In: *Communications of the ACM* 5.7 (1962), pp. 394–397.

[SS96] Joao P Marques Silva and Karem A Sakallah. “GRASP-a new search algorithm for satisfiability.” In: *ICCAD*. Vol. 96. 1996, pp. 220–227.

[CDE+08] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. “Klee: unassisted and automatic generation of high-coverage tests for complex systems programs.” In: *OSDI*. Vol. 8. 2008, pp. 209–224.

[Sho+16] Yan Shoshitaishvili et al. “SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis”. In: *IEEE S&P ’16*. 2016.

[GLM12] Patrice Godefroid, Michael Y Levin, and David Molnar. “SAGE: whitebox fuzzing for security testing”. In: *Communications of the ACM* 55.3 (2012), pp. 40–44.

[Sto76] Larry J. Stockmeyer. “The polynomial-time hierarchy”. In: *TCS ’76* 3.1 (1976), pp. 1–22. DOI: https://doi.org/10.1016/0304-3975(76)90061-X. URL: https://www.sciencedirect.com/science/article/pii/030439757690061X.

[CMV16b] Supratik Chakraborty, Kuldeep S Meel, and Moshe Y Vardi. *Algorithmic improvements in approximate counting for probabilistic inference: From linear to logarithmic SAT calls*. Tech. rep. National University of Singapore, 2016.

[Val79a] Leslie G Valiant. “The complexity of computing the permanent”. In: *Theoretical computer science* 8.2 (1979), pp. 189–201.

[Cha+14b] Supratik Chakraborty et al. *Distribution-Aware Sampling and Weighted Model Counting for SAT*. 2014. arXiv: 1404.2984 [cs.AI].

[Ras+13] Aseem Rastogi et al. “Knowledge inference for optimizing secure multi-party computation”. In: *ACM SIGPLAN workshop on Programming languages and analysis for security*. 2013, pp. 3–14.

[Alm+18] José Bacelar Almeida et al. “Enforcing ideal-world leakage bounds in real-world secret sharing MPC frameworks”. In: *IEEE CSF ’18*. 2018, pp. 132–146.

[Buh+22] Ileana Buhan et al. “SoK: Design tools for side-channel-aware implementations”. In: *ACM AsiaCCS ’22*. 2022, pp. 756–770.

[WHK14] Dan Walters, Andrew Hagen, and Eric Kedaigle. *Sleak: A side-channel leakage evaluator and analysis kit*. Tech. rep. MITRE CORP BEDFORD MA, 2014.

[Lap+16] Andrei Lapets et al. “Secure MPC for analytics as a web application”. In: *IEEE SecDev ’16*. 2016, pp. 73–74.

[Bün+18] Benedikt Bünz et al. “Bulletproofs: Short proofs for confidential transactions and more”. In: *IEEE S&P ’18*. IEEE. 2018, pp. 315–334.

[Sas+14] Eli Ben Sasson et al. “Zerocash: Decentralized anonymous payments from bitcoin”. In: *IEEE S&P ’14*. 2014, pp. 459–474.

[Bog+09] Peter Bogetoft et al. “Secure multiparty computation goes live”. In: *International Conference on Financial Cryptography and Data Security*. 2009, pp. 325–343.

[Pyt22] Python Software Foundation. *Process Pools*. 2022. URL: https://docs.python.org/3/library/multiprocessing.html#module-multiprocessing.pool.

[Sha] Arijit Shaw. *Research Statement*. https://www.tcgcrest.org/wp-content/uploads/2022/02/Arijit.pdf. Private communications with author.

[Joi+24] Tushar M Jois et al. "SocIoTy: Practical Cryptography in Smart Home Contexts". In: *Proceedings on Privacy Enhancing Technologies* (2024).

[Tos+12] Chad Tossell et al. "Characterizing web use on smartphones". In: *Proceedings of the SIGCHI conference on human factors in computing systems*. 2012, pp. 2769–2778.

[AK12] George Argyros and Aggelos Kiayias. "I forgot your password: Randomness attacks against {PHP} applications". In: *21st USENIX Security Symposium (USENIX Security 12)*. 2012, pp. 81–96.

[ZJG22] Maximilian Zinkus, Tushar M Jois, and Matthew Green. "SoK: Cryptographic Confidentiality of Data on Mobile Devices". In: *Proceedings on Privacy Enhancing Technologies* 2022.1 (2022), pp. 586–607.

[Pew21] Pew Research Center. *Mobile Fact Sheet*. Accessed 7/29/2022. 2021. URL: `\url{https://www.pewresearch.org/internet/fact-sheet/mobile/}`.

[Cha+09] N. Chandran et al. "Position Based Cryptography". In: *CRYPTO 2009*. Ed. by Shai Halevi. Vol. 5677. LNCS. Springer, Heidelberg, 2009, pp. 391–407. DOI: 10.1007/978-3-642-03356-8_23.

[Bro+17] Joshua Brody et al. "Position-Based Cryptography and Multiparty Communication Complexity". In: *TCC 2017, Part I*. Ed. by Yael Kalai and Leonid Reyzin. Vol. 10677. LNCS. Springer, Heidelberg, 2017, pp. 56–81. DOI: 10.1007/978-3-319-70500-2_3.

[Buh+11] Harry Buhrman et al. "Position-Based Quantum Cryptography: Impossibility and Constructions". In: *CRYPTO 2011*. Ed. by Phillip Rogaway. Vol. 6841. LNCS. Springer, Heidelberg, 2011, pp. 429–446. DOI: 10.1007/978-3-642-22792-9_24.

[Phu+19] Tran Viet Xuan Phuong et al. "Location Based Encryption". In: *ACISP 19*. Ed. by Julian Jang-Jaccard and Fuchun Guo. Vol. 11547. LNCS. Springer, Heidelberg, 2019, pp. 21–38. DOI: 10.1007/978-3-030-21548-4_2.

[PQ10] Kenneth G Paterson and Elizabeth A Quaglia. "Time-specific encryption". In: *International Conference on Security and Cryptography for Networks*. Springer. 2010, pp. 1–16.

[Kas+12] Kohei Kasamatsu et al. "Time-specific encryption from forward-secure encryption". In: *International Conference on Security and Cryptography for Networks*. Springer. 2012, pp. 184–204.

[SD03] 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*. 2003, pp. 734–740.

[Qiu+07] Di Qiu et al. “Geoencryption using loran”. In: *Proceedings of the 2007 National Technical Meeting of The Institute of Navigation*. 2007, pp. 104–115.

[FV12] M. D. Firoozjaei and J. Vahidi. “Implementing geo-encryption in GSM cellular network”. In: *2012 9th International Conference on Communications (COMM)*. IEEE. 2012, pp. 299–302.

[AFAI07] Ala Al-Fuqaha and Omar Al-Ibrahim. “Geo-encryption protocol for mobile networks”. In: *Computer Communications* 30.11-12 (2007), pp. 2510–2517.

[SRB20] Jayasree Sengupta, Sushmita Ruj, and Sipra Das Bit. “A comprehensive survey on attacks, security issues and blockchain solutions for IoT and IIoT”. In: *Journal of Network and Computer Applications* 149 (2020), p. 102481.

[DV17] 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)*. IEEE. 2017, pp. 32–37.

[Mug18] Arif Ali Mughal. “The Art of Cybersecurity: Defense in Depth Strategy for Robust Protection”. In: *International Journal of Intelligent Automation and Computing* 1.1 (2018), pp. 1–20.

[Yao86b] Andrew Chi-Chih Yao. “How to Generate and Exchange Secrets (Extended Abstract)”. In: *27th FOCS*. IEEE Computer Society Press, 1986, pp. 162–167. DOI: 10.1109/SFCS.1986.25.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. “How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority”. In: *19th ACM STOC*. Ed. by Alfred Aho. ACM Press, 1987, pp. 218–229. DOI: 10.1145/28395.28420.

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. “Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation (Extended Abstract)”. In: *20th ACM STOC*. ACM Press, 1988, pp. 1–10. DOI: 10.1145/62212.62213.

[CCD88] David Chaum, Claude Crépeau, and Ivan Damgård. “Multiparty Unconditionally Secure Protocols (Abstract) (Informal Contribution)”. In: *CRYPTO’87*. Ed. by Carl Pomerance. Vol. 293. LNCS. Springer, Heidelberg, 1988, p. 462. DOI: 10.1007/3-540-48184-2_43.

[AS+17] Shadi Al-Sarawi et al. “Internet of Things (IoT) communication protocols”. In: *2017 8th International conference on information technology (ICIT)*. IEEE. 2017, pp. 685–690.

[Cho+21] Arka Rai Choudhuri et al. “Fluid MPC: Secure Multiparty Computation with Dynamic Participants”. In: *CRYPTO 2021, Part II*. Ed. by Tal Malkin and Chris Peikert. Vol. 12826. LNCS. Virtual Event: Springer, Heidelberg, 2021, pp. 94–123. DOI: 10.1007/978-3-030-84245-1_4.

[Tya+18] Nirvan Tyagi et al. “BurnBox: Self-Revocable Encryption in a World Of Compelled Access”. In: *USENIX Security 2018*. Ed. by William Enck and Adrienne Porter Felt. USENIX Association, 2018, pp. 445–461.

[M’R+11] David M’Raihi et al. *RFC 6238: TOTP: Time-based one-time password algorithm*. 2011.

[Mor22] Mordor Intelligence. *Global smart homes market—growth, analysis, forecast to 2022*. 2022. URL: \url{https://www.mordorintelligence.com/industry-reports/global-smart-homes-market-industry}.

[Sta22] Statista. *Digital market - Smart Home*. 2022. URL: https://www.statista.com/outlook/dmo/smart-home/worldwide#smart-homes.

[Kum+19] Sam Kumar et al. “JEDI: Many-to-Many End-to-End Encryption and Key Delegation for IoT”. In: *28th USENIX Security Symposium (USENIX Security 19)*. 2019, pp. 1519–1536.

[Che21] Chrysta Cherrie. *The 2021 State of the Auth Report: 2FA Climbs, While Password Managers and Biometrics Trend*. Accessed 7/29/2022. 2021. URL: https://duo.com/blog/the-2021-state-of-the-auth-report-2fa-climbs-password-managers-biometrics-trend.

[Wat21] Yash Wate. *How to Enable Two-Factor Authentication on Facebook, Instagram, and Twitter*. Accessed 7/29/2022. 2021. URL: https://techpp.com/2020/02/03/enable-two-factor-authentication-instagram-facebook-twitter/.

[Duo21] Duo. *Duo Authentication for Epic*. Accessed 7/29/2022. 2021. URL: https://duo.com/docs/epic.

[Rob] Robinhood. *Two-Factor Authentication*. Accessed 7/29/2022. URL: https://robinhood.com/us/en/support/articles/twofactor-authentication/.

[Cis20] Cisco. *Configure AnyConnect Secure Mobility Client using One-Time Password (OTP) for Two-Factor Authentication on an ASA*. Accessed 7/29/2022. 2020. URL: https://www.cisco.com/c/en/us/support/docs/security/anyconnect-secure-mobility-client/213931-configure-anyconnect-secure-mobility-cli.html.

[M’R+05] David M’Raihi et al. *RFC 4226: HOTP: An hmac-based one-time password algorithm*. 2005.

[KBC97] Hugo Krawczyk, Mihir Bellare, and Ran Canetti. *RFC 2104: HMAC: Keyed-hashing for message authentication*. 1997.

[GGM86] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. “How to construct random functions”. In: *Journal of the ACM (JACM)* 33.4 (1986), pp. 792–807.

[SV21] Sarah Scheffler and Mayank Varia. “Protecting Cryptography Against Compelled {Self-Incrimination}”. In: *30th USENIX Security Symposium (USENIX Security 21)*. 2021, pp. 591–608.

[GRR98] Rosario Gennaro, Michael O. Rabin, and Tal Rabin. “Simplified VSS and Fast-Track Multiparty Computations with Applications to Threshold Cryptography”. In: *17th ACM PODC*. Ed. by Brian A. Coan and Yehuda Afek. ACM, 1998, pp. 101–111. DOI: 10.1145/277697.277716.

[Chi+18] Koji Chida et al. “Fast Large-Scale Honest-Majority MPC for Malicious Adversaries”. In: *CRYPTO 2018, Part III*. Ed. by Hovav Shacham and Alexandra Boldyreva. Vol. 10993. LNCS. Springer, Heidelberg, 2018, pp. 34–64. DOI: 10.1007/978-3-319-96878-0_2.

[RS21] Rahul Rachuri and Peter Scholl. *Le Mans: Dynamic and Fluid MPC for Dishonest Majority*. Cryptology ePrint Archive, Report 2021/1579. https://eprint.iacr.org/2021/1579. 2021.

[Gen+21] Craig Gentry et al. “YOSO: You Only Speak Once - Secure MPC with Stateless Ephemeral Roles”. In: *CRYPTO 2021, Part II*. Ed. by Tal Malkin and Chris Peikert. Vol. 12826. LNCS. Virtual Event: Springer, Heidelberg, 2021, pp. 64–93. DOI: 10.1007/978-3-030-84245-1_3.

[KRY22] Sebastian Kolby, Divya Ravi, and Sophia Yakoubov. *Towards Efficient YOSO MPC Without Setup*. Cryptology ePrint Archive, Report 2022/187. https://eprint.iacr.org/2022/187. 2022.

[Cas+22] Ignacio Cascudo et al. *YOLO YOSO: Fast and Simple Encryption and Secret Sharing in the YOSO Model*. Cryptology ePrint Archive, Report 2022/242. https://eprint.iacr.org/2022/242. 2022.

[Res18] E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. [Online; accessed 29. Jul. 2022]. 2018. DOI: 10.17487/RFC8446.

[ECM13] ECMA International. *ECMA-340 Near Field Communication Interface and Protocol (NFCIP-1)*. 2013.

[Nor] Nordic Semiconductor. *Near Field Communication (NFC)*. URL: https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/nrf/ug_nfc.html.

[Bar+20] Manuel Barbosa et al. *Provable Security Analysis of FIDO2*. Cryptology ePrint Archive, Report 2020/756. https://eprint.iacr.org/2020/756. 2020.

[Alb+16] Martin R. Albrecht et al. “MiMC: Efficient Encryption and Cryptographic Hashing with Minimal Multiplicative Complexity”. In: *ASIACRYPT 2016, Part I*. Ed. by Jung Hee Cheon and Tsuyoshi Takagi. Vol. 10031. LNCS. Springer, Heidelberg, 2016, pp. 191–219. DOI: 10.1007/978-3-662-53887-6_7.

[Kel+17] Marcel Keller et al. “Faster Secure Multi-party Computation of AES and DES Using Lookup Tables”. In: *ACNS 17*. Ed. by Dieter Gollmann, Atsuko Miyaji, and Hiroaki Kikuchi. Vol. 10355. LNCS. Springer, Heidelberg, 2017, pp. 229–249. DOI: 10.1007/978-3-319-61204-1_12.

[DK10] Ivan Damgård and Marcel Keller. “Secure multiparty AES”. In: *International Conference on Financial Cryptography and Data Security*. Springer. 2010, pp. 367–374.

[Alb+15] M. Albrecht et al. “Ciphers for MPC and FHE”. In: *EUROCRYPT 2015, Part I*. Ed. by Elisabeth Oswald and Marc Fischlin. Vol. 9056. LNCS. Springer, Heidelberg, 2015, pp. 430–454. DOI: 10.1007/978-3-662-46800-5_17.

[AD18] Tomer Ashur and Siemen Dhooghe. *MARVELlous: a STARK-Friendly Family of Cryptographic Primitives*. Cryptology ePrint Archive, Report 2018/1098. https://eprint.iacr.org/2018/1098. 2018.

[Aly+20] Abdelrahaman Aly et al. “Design of Symmetric-Key Primitives for Advanced Cryptographic Protocols”. In: *IACR Trans. Symm. Cryptol.* 2020.3 (2020), pp. 1–45. ISSN: 2519-173X. DOI: 10.13154/tosc.v2020.i3.1-45.

[Gra+21] Lorenzo Grassi et al. “Poseidon: A New Hash Function for Zero-Knowledge Proof Systems”. In: *USENIX Security 2021*. Ed. by Michael Bailey and Rachel Greenstadt. USENIX Association, 2021, pp. 519–535.

[Arc+] David Archer et al. *Bristol Fashion MPC Circuits*. URL: https://homes.esat.kuleuven.be/~nsmart/MPC/.

[Hal+98] Chris Hall et al. “Building PRFs from PRPs”. In: *CRYPTO’98*. Ed. by Hugo Krawczyk. Vol. 1462. LNCS. Springer, Heidelberg, 1998, pp. 370–389. DOI: 10.1007/BFb0055742.

[BI99] M. Bellare and R. Impagliazzo. *A tool for obtaining tighter security analyses of pseudorandom function based constructions, with applications to PRP to PRF conversion*. Cryptology ePrint Archive, Report 1999/024. https://eprint.iacr.org/1999/024. 1999.

[GGM18] Shoni Gilboa, Shay Gueron, and Ben Morris. “How many queries are needed to distinguish a truncated random permutation from a random function?” In: *Journal of Cryptology* 31.1 (2018), pp. 162–171.

[Kha+19] Md Sakib Nizam Khan et al. “chownIoT: Enhancing IoT Privacy by Automated Handling of Ownership Change”. In: *Privacy and Identity Management. Fairness, Accountability, and Transparency in the Age of Big Data: 13th IFIP WG 9.2, 9.6/11.7, 11.6/SIG 9.2.2 International Summer School, Vienna, Austria, August 20-24, 2018, Revised Selected Papers*. Ed. by Eleni Kosta et al. Cham: Springer International Publishing, 2019, pp. 205–221. ISBN: 978-3-030-16744-8. DOI: 10.1007/978-3-030-16744-8_14.

[GG22] Martin Gunnarsson and Christian Gehrmann. “Secure Ownership Transfer for Resource Constrained IoT Infrastructures”. In: *Information Systems Security and Privacy*. Ed. by Steven Furnell et al. Cham: Springer International Publishing, 2022, pp. 22–47. ISBN: 978-3-030-94900-6. DOI: 10.1007/978-3-030-94900-6_2.

[WZZ93] Xingwei Wang, Hong Zhao, and Jiakeng Zhu. “GRPC: A communication cooperation mechanism in distributed systems”. In: *ACM SIGOPS Operating Systems Review* 27.3 (1993), pp. 75–86.

[Can+03b] Brice Canvel et al. “Password Interception in a SSL/TLS Channel”. In: *CRYPTO 2003*. Ed. by Dan Boneh. Vol. 2729. LNCS. Springer, Heidelberg, 2003, pp. 583–599. DOI: 10.1007/978-3-540-45146-4_34.

[Mey+14] Christopher Meyer et al. “Revisiting SSL/TLS Implementations: New Bleichenbacher Side Channels and Attacks”. In: *USENIX Security 2014*. Ed. by Kevin Fu and Jaeyeon Jung. USENIX Association, 2014, pp. 733–748.

[Gar+16b] Christina Garman et al. “Dancing on the Lip of the Volcano: Chosen Ciphertext Attacks on Apple iMessage”. In: *USENIX Security 2016*. Ed. by Thorsten Holz and Stefan Savage. USENIX Association, 2016, pp. 655–672.

[BSY18] Hanno Böck, Juraj Somorovsky, and Craig Young. “Return Of Bleichenbacher’s Oracle Threat (ROBOT)”. In: *USENIX Security 2018*. Ed. by William Enck and Adrienne Porter Felt. USENIX Association, 2018, pp. 817–849.

[BZG20b] Gabrielle Beck, Maximilian Zinkus, and Matthew Green. “Automating the Development of Chosen Ciphertext Attacks”. In: *USENIX Security 2020*. Ed. by Srdjan Capkun and Franziska Roesner. USENIX Association, 2020, pp. 1821–1837.

[LGR21] Julia Len, Paul Grubbs, and Thomas Ristenpart. "Partitioning Oracle Attacks". In: *USENIX Security 2021*. Ed. by Michael Bailey and Rachel Greenstadt. USENIX Association, 2021, pp. 195–212.

[BHP22] Matilda Backendal, Miro Haller, and Kenneth G. Paterson. *MEGA: Malleable Encryption Goes Awry*. 2022. URL: https://eprint.iacr.org/2022/959.

[RH22] Keegan Ryan and Nadia Heninger. *Cryptanalyzing MEGA in Six Queries*. 2022. URL: https://eprint.iacr.org/2022/914.

[WG11] Marko Wolf and Timo Gendrullis. "Design, implementation, and evaluation of a vehicular hardware security module". In: *International Conference on Information Security and Cryptology*. Springer. 2011, pp. 302–318.

[Hup+20] William Hupp et al. "Module-OT: a hardware security module for operational technology". In: *2020 IEEE Texas Power and Energy Conference (TPEC)*. IEEE. 2020, pp. 1–6.

[Han+19] Juhyeng Han et al. "Toward scaling hardware security module for emerging cloud services". In: *Proceedings of the 4th Workshop on System Software for Trusted Execution*. 2019, pp. 1–6.

[Cao+20] Yetong Cao et al. "PPGPass: Nonintrusive and secure mobile two-factor authentication via wearables". In: *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE. 2020, pp. 1917–1926.

[Chu14] John Chuang. "One-step two-factor authentication with wearable biosensors". In: *Symposium on Usable Privacy and Security-SOUPS*. Vol. 14. 2014.

[SS18] 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*. 2018, pp. 99–110.

[Zha+17] Jiansong Zhang et al. "Proximity based IoT device authentication". In: *IEEE INFOCOM 2017-IEEE conference on computer communications*. IEEE. 2017, pp. 1–9.

[ABS18] Muhammad Naveed Aman, Mohamed Haroon Basheer, and Biplab Sikdar. "Two-factor authentication for IoT with location information". In: *IEEE Internet of Things Journal* 6.2 (2018), pp. 3335–3351.

[Kas18] Anand Kashyap. *The Next Generation: HSM approach delivers unparalleld cost/benefit for organizations*. 2018. URL: https://securitytoday.com/Articles/2018/12/01/The-Next-Generation.aspx.