Automating the Development of Chosen Ciphertext Attacks

Gabrielle Beck[*]
Johns Hopkins University
becgabri@cs.jhu.edu

Maximilian Zinkus[*]
Johns Hopkins University
zinkus@cs.jhu.edu

Matthew Green
Johns Hopkins University
mgreen@cs.jhu.edu

Abstract

In this work we investigate the problem of automating the development of adaptive chosen ciphertext attacks on systems that contain vulnerable *format oracles*. Unlike previous attempts, which simply automate the execution of known attacks, we consider a more challenging problem: to programmatically derive a novel attack strategy, given only a machine-readable description of the plaintext verification function and the malleability characteristics of the encryption scheme. We present a new set of algorithms that use SAT and SMT solvers to reason deeply over the design of the system, producing an automated attack strategy that can entirely decrypt protected messages. Developing our algorithms required us to adapt techniques from a diverse range of research fields, as well as to explore and develop new ones. We implement our algorithms using existing theory solvers. The result is a practical tool called `Delphinium` that succeeds against real-world and contrived format oracles. To our knowledge, this is the first work to automatically derive such complex chosen ciphertext attacks.

1 Introduction

The past decades have seen enormous improvement in our understanding of cryptographic 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's CBC-mode ciphers [Smi12, AP13], export-grade encryption [BBDL$^+$15, ABD$^+$15, ASS$^+$16], and legacy email encryption [PDM$^+$18]. However, support for legacy protocols does not account for the presence of vulnerabilities in more recent protocols and systems [W3C17, JS11, MRLG15, GGK$^+$16, VP17].

In this work 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, BFK$^+$12, AP13, MDK14, APW09, DP10, CHVV03, Mit05]. Despite repeated warnings to industry, variants of these attacks continue to plague modern systems, including TLS 1.2's CBC-mode ciphersuite [AP13, AP15, MSA$^+$19] and hardware key management tokens [BFK$^+$12, AF18]. A generalized variant, the *format oracle attack* can be constructed when a decryption oracle leaks the result of applying some (arbitrarily complex) format-checking predicate F to a decrypted plaintext. Format oracles appear even in recent standards such as XML encryption [JS11, KMSS15], Apple's iMessage [GGK$^+$16] and modern OpenPGP implementations [MRLG15, PDM$^+$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.

[*] These authors contributed equally to the work.

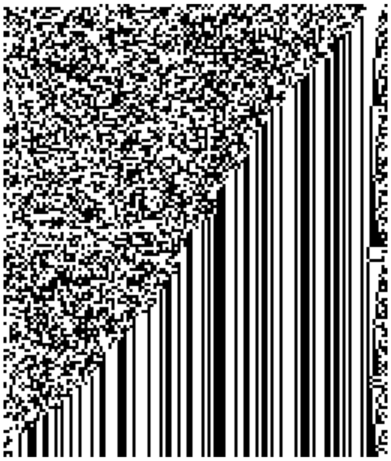


Figure 1: Output of a format oracle attack that our algorithms developed against a bitwise padding check oracle $\mathsf{F}_{\mathsf{bitpad}}$ (see §5.2 for a full description). The original ciphertext is a valid 128-bit (random) padded message encrypted using a stream cipher. Each row of the bitmap represents a *malleation string* that was exclusive-ORed with the ciphertext prior to making a decryption query.

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, BZD+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 oracles which require entirely new exploit strategies. As a concrete example, the authors of [GGK+16] 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 [GGK+16, PDM+18] and PDF encryption [MIM+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 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 [KMSS15]. 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.

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 of this work is a toolset that we name `Delphinium`, which produces highly-efficient attacks across several such functions.

Relationship to previous automated attack work. Previous work [BRB18, PBP$^+$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. In this work we make the following contributions:

- 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.

- 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.

- 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.

- We develop a working implementation of our techniques using "off-the-shelf" SAT/SMT packages, and provide the resulting software package (which we call `Delphinium`), an artifact accompanying this submission, as an open source tool for use and further development by the research community.

- 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.

1.1 Intuition

We now describe our problem setting and provide an overview of the techniques that we use in this work. *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.

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 $\mathsf{Maul}^{\Pi}_{\mathsf{ciph}}(C,S) \to C'$ 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 $\mathsf{Maul}^{\Pi}_{\mathsf{plain}}(M,S) \to M'$ computes the equivalent malleation over some plaintext, producing a plaintext (or, in some cases, a set of possible plaintexts[1]). 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) [stp, dMB08]. 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.

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.

In our setting, each experiment comprises a concrete malleation string $\mathbf{S}$ that can be applied to the target ciphertext C^* using the ciphertext malleation function. The technical challenge is to programmatically derive such a malleation string that (a) represents a meaningful input to the malleation functions, and (b) ensures

[1] 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.

that the results of an experiment will be *useful*, *i.e.,* that the attack will make *progress* by eliminating some candidate plaintexts.

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^*.[2] (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:

$$\begin{gathered} G_{i-1}(M_0) = G_{i-1}(M_1) = 1 \ \wedge \\ \forall b \in \{0,1\} : \mathsf{F}(\mathsf{Maul}_{\mathsf{plain}}(M_b, S)) = b \end{gathered} \tag{1}$$

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).

[2] Here n represents an upper bound on the length of the plaintext M^*.

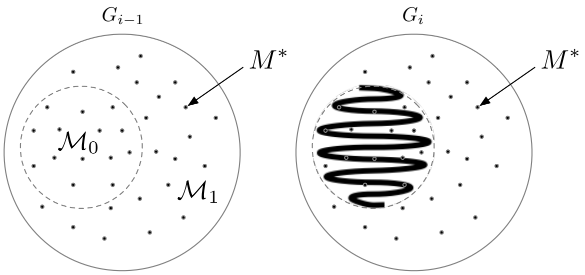


Figure 2: Left: illustration of a plaintext candidate space defined by G_{i-1}, highlighting the two subsets $\mathcal{M}_0, \mathcal{M}_1$ induced by a specific malleation string $\mathbf{S}$. Right: the candidate space defined by G_i, in which many candidates have been eliminated following an oracle response $b = 1$.

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, 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. 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 [dMB08], 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).[3]

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 [Val79, 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 $\mathsf{NP}^{\mathsf{PP}}$-complete [FRS17, Tod91], several works have explored *approximate* solutions to this and related counting problems [GSS06, CMV16, 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 $H_j : \{0,1\}^{|T|} \to \{0,1\}$ 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 assigment 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 to our attack, 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.[4] 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.

[3] 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.

[4] 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.

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 iff 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 §5.

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 §4.

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 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.

2.1 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.

2.1.1 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 $\mathsf{Maul}^{\Pi}_{\mathsf{ciph}} : \mathcal{C} \times \{0,1\}^* \rightarrow \mathcal{C} \cup \{\perp\}$ takes as input a ciphertext and a *malleation instruction string*. It outputs a ciphertext or the distinguished error symbol $\perp$. The function $\mathsf{Maul}^{\Pi}_{\mathsf{plain}} : \mathcal{M} \times \{0,1\}^* \rightarrow \hat{\mathcal{M}}$, 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.

A malleation function pair must correctly represent the effect of mauling ciphertexts produced by the encryption scheme. This is captured by the following notion:

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). We define the functions $\mathsf{Maul}^{\Pi_{\mathsf{stream}}}_{\mathsf{plain}}$ and $\mathsf{Maul}^{\Pi_{\mathsf{stream}}}_{\mathsf{ciph}}$ 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.[5] We can define a malleation function $\mathsf{Maul}^{\Pi_{\mathsf{tstream}}}_{\mathsf{ciph}}$ (resp. $\mathsf{Maul}^{\Pi_{\mathsf{tstream}}}_{\mathsf{plain}}$) 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 discuss these in §4.

Remarks. Note that the function $\mathsf{Maul}^{\Pi}_{\mathsf{ciph}}$ outputs a single ciphertext. By contrast, the plaintext malleation function $\mathsf{Maul}^{\Pi}_{\mathsf{plain}}$ outputs a *set of possible decryption results*. This captures the possibility that certain forms

[5] 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.

of malleation can induce unpredictable key-dependent effects when ciphertexts are decrypted.[6] 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 $\mathcal{M} \cup \{\bot\}$. 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).

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, $i = 0$). To describe this scheme, 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 §4 we describe a more efficient encoding of this output.

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, DLL62] 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. 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.[7] 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. 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 $\mathsf{unknown}$.

Model counting and $\mathsf{Max\#SAT}$**.** While SAT determines the existence of a single satisfying assignment, a more general variant of the problem, $\mathsf{\#SAT}$, determines the *number* of satisfying assignments. [8] In the literature this problem is known as *model counting* [Val79, GSS06, BL99, BJP00, SBB^{+}04, WS05, BDP03, CFMV15].

In this work we make use of a specific optimization variant of the model count problem, which was formulated as $\mathsf{Max\#SAT}$ 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.[9] We will make use of this abstraction in

[6] 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.

[7] In principle our attacks can be extended to other theories, with some additional work that we describe later in this section.

[8] 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 easily be viewed as a slightly modified version of $\mathsf{\#SAT}$.

[9] The formulation of Fremont *et al.* [FRS17] 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.

our attacks, with realizations discussed in §3.2. Specifically, we define our main attack algorithm in terms of a generic Max#SAT oracle that has the following interface:

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

Fremont *et al.* provide an algorithm called MaxCount that approximately solves the Max#SAT problem, using several approximate sampling and counting algorithms [CMV16, CFM$^+$14, CFMV15] as an ingredient. 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.

2.3 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 [dMB08] 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.* [PBP$^+$17] demonstrated techniques for deriving such constraint systems automatically from high-level languages such as Java which could be combined with our work.

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`.

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 §1.1. At each iteration, it derives a concrete malleation string **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 O_{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.[10]

Theorem 3.1 *Given an exact* Max#SAT *oracle, Algorithm 1 maximizes in expectation the number of candidate plaintext messages ruled out at each iteration.*

A proof of Theorem 3.1 appears in Appendix A.

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 [PBP+17]). We find that this is computationally infeasible in practice. Finally, note also that our proof assumes an exact Max#SAT oracle. In practice, this will likely be realized with a probably approximately correct instantiation, causing the resulting attack to be a probably approximately greedy attack. Appendices B and C describe example formats for which the greedy attack is provably optimal and non-optimal, respectively.

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 [CMV16, CFM+14, CFMV15], 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 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 **S** that (heuristically) maximizes the number of solutions for the bitvectors M_0, M_1. It does this by first

[10] Our concrete implementation in §4 uses the solver to enumerate each of the known and unknown bits of M^*.

conjecturing some value s, and sampling a series of $2s$ low-density parity hash functions of the form $H : \{0,1\}^n \rightarrow \{0,1\}$ (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 §5.

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.

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

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

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, $b_{\mathsf{continue}} = \text{TRUE}$, and $Z' = Z$.

Finally, the subroutine ParityConstraint(n, l) constructs l randomized parity constraints of weight k over a bitvector $b = b_1 b_2 \ldots b_n$ 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).

3.3 Attacks with Knowledge of the Plaintext Distribution

The attack as presented in the previous section takes an initial constraint formula G_0 that represents the attacker's initial knowledge of the structure of M^*. This formula can be left empty, however in a typical case where the ciphertext contains a known-valid message (*i.e.,* $\mathsf{F}(M^*) = 1$), the user can set $G_0 \leftarrow \mathsf{F}$. In some cases the user may know significantly more detailed information about the distribution of M^*: for example, that it uses a specific format such as *e.g.,* UTF-7 or JSON. This sort of auxiliary data can be encoded into G_0 in order to improve the runtime of the attack.

Optimization: weighting the plaintext distribution. 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 assumes that every valid plaintext admitted by G_i is equally likely, and thus the most efficient path to finding M^* is to simply maximize the number of plaintext candidates eliminated. This may not produce an optimal attack. In Appendix D we discuss a more nuanced approach that allows the user to *weight* plaintexts by their probability of occurring in 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 $\big[G_{i-1}(M_0) = 1 \wedge G_{i-1}(M_1) = 1 \wedge \quad \mathsf{F}(\mathsf{Maul}_{\mathsf{plain}}(M_0, S)) = 0 \wedge$
        $\mathsf{F}(\mathsf{Maul}_{\mathsf{plain}}(M_1, S)) = 1\big]$;
     $\mathbf{S} \leftarrow$ Max#SAT ($\phi$, $S$, $M_0 \| M_1$);
     if $S \neq \perp$ then
        $\mathbf{r} \leftarrow O_{\mathsf{dec}}(\mathsf{Maul}_{\mathsf{ciph}}(C^*, \mathbf{S}))$;
        Define $G_i(M)$ as $\big[G_{i-1}(M) \wedge (\mathsf{F}(\mathsf{Maul}_{\mathsf{plain}}(M, \mathbf{S})) = \mathbf{r})\big]$;
        $i \leftarrow i + 1$;
  while $S \neq \perp$;
  return SolveForPlaintext($G_i$);
```

```
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:
  $(b_{\mathsf{continue}}, s, Z) \leftarrow$ AdjustSize($\textsc{False}, n, \perp, \varepsilon$);
  // 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 $b_{\mathsf{continue}}$ 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
     $\mathbf{S} \leftarrow \mathsf{SATSolve}\{(S, \{M_{1,0}, \ldots, M_{t,0}\}, \{M_{1,1}, \ldots, M_{t,1}\})$ :
           $\exists \mathcal{R}_0 \subseteq [1,t] : |\mathcal{R}_0| \geq \lceil (0.5 + \delta)t \rceil, \forall j \in \mathcal{R}_0 : \mathcal{H}_{j,0}(M_{j,0}) = 1 \wedge$
           $\exists \mathcal{R}_1 \subseteq [1,t] : |\mathcal{R}_1| \geq \lceil (0.5 + \delta)t \rceil, \forall j \in \mathcal{R}_1 : \mathcal{H}_{j,1}(M_{j,1}) = 1 \wedge$
           $\phi^t(S, \{M_{1,0}, \ldots, M_{t,0}\}, \{M_{1,1}, \ldots, M_{t,1}\})\}$;
        if $S == \mathsf{unsat}$ then
              $b_{\mathsf{success}} = \textsc{False}$;
        $(b_{\mathsf{continue}}, s, Z) \leftarrow$ AdjustSize($b_{\mathsf{success}}, n, s, Z$);
  return $\mathbf{S}$
```

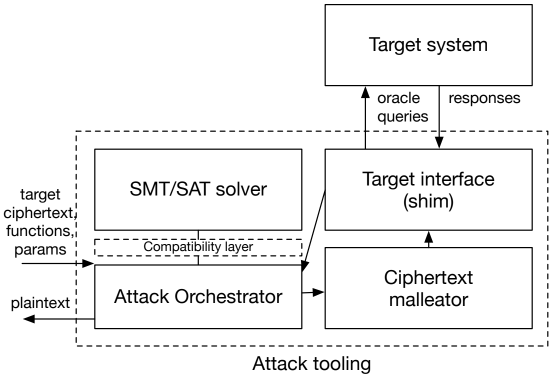


Figure 3: Architecture of `Delphinium`.

4 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.

4.1 Architecture Overview

Figure 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 SMT solver frameworks (STP [stp] and Z3 [dMB08]) via a custom compatibility layer that we developed for our tool. To improve performance, the orchestrator may launch multiple parallel instances of this solver.

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.[11]

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.

4.2 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.

[11] The interface requires input of a ciphertext and a malleation string, with output the mauled ciphertext.

Test Harness. For our experiments in §5 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 **S**.

Selecting SAT and SMT solvers. In the course of this work we evaluated several SMT and SAT solvers, including Z3 [dMB08], Boolector [NPWB18], STP [stp] and CryptoMiniSAT [SNC09] (henceforth CMS). In 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.

We determined that each package offers significant performance and functionality tradeoffs. Z3 has the most complete feature set, tooling, and support, and thus was our starting point. Unfortunately, Z3 has one critical *disadvantage*: the default SAT solver (MiniSAT [min]) does not by default perform Gaussian elimination of exclusive-OR gates. The conversion required to handle these gates produces an exponential increase in formula complexity, which renders our model counting techniques intractable even at modest parameters. Boolector is optimized for bitvector operations, but was significantly less usable due to software issues particularly pertaining to the Python bindings.

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 a compromise, 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.[12] 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.

A note on future solver development. The complexity of our constraint formulae uncovered a number of interesting software bugs in the SMT/SAT solver packages we worked with. In every case, we received excellent support from the development teams. However, a corrollary of this result is that our techniques may benefit from significant further optimization by the theory solver research and development community. We are hopeful that the experimental results presented in this work represent a baseline, and that publication of work will motivate further performance improvements.

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

[12] As future work, we are working to develop native Z3 integration for CMS, which will likely provide substantially improved performance.

works [ZCSE16, EGSS14]: 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. 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.

To avoid this runtime overhead, we employ a parallelization strategy to explore many close values of s simultaneously. To do this, we create a distinct solver instance for each of a "window" of sequential values $(s_1, \ldots, s_N)$ and execute each of these instances in parallel, selecting the largest satisfied result.[13]

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 making users re-implement 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.

4.2.1 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

[13] To avoid the overhead of a `fork` system call, we arrived at a hybrid implementation in which a Z3 solver instance in one process exports its representation of the constraints as CNF, which we replicate and pass to many parallel instances of CMS after adding the appropriate parity constraints for each subset size s. The benefit of this approach is that it avoids performance bottlenecks in CNF generation caused by exclusive-OR expansion.

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.[14] 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.[15] 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.

4.3 Implementation Considerations

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

[14] This is not necessarily possible when dealing with other stream ciphers, due to the keystream being misaligned with the ciphertext.

[15] 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.

is also highly parallelizable: the inner loop of FastSample can be run with many different values for s simultaneously. We refer to the width of the range of values for s 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. Experientially it is faster to run larger windows with fewer cores per solver.

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.

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. We export CNF *excluding* the exclusive-ORs required by the model counting algorithm, and then append them to the CNF output. CMS is able to process such supplemented CNF (CNFXOR) files efficiently, with exclusive-ORs represented in linearly-sized statements.

4.4 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.

4.5 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 [MGC+16, FHK+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, SRS+18]. We additionally provide a translation tool from the output format of CMBC-GC [FHK+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 [WBL$^+$19] may provide an additional avenue for format function constraint generation in future work.

5 Experiments

5.1 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 and 144GB of RAM.[16] 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 §5.2. Empirically, $\delta = 0.5$, $2 \leqslant t \leqslant 5$, and parity functions of logarthmic length are suitable for our purposes. Experiments varying t and comparing parity hash function lengths can be found in Appendix E. These tests were performed on AWS.

5.2 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. For our initial experiments we do not consider truncation or other forms of editing. We used these ciphers to implement several format oracles as described below.

Bytewise Encryption Padding. The PKCS #7 encryption standard (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. Throughout the rest of this paper, we refer to these schemes as PKCS7 and TLS-PKCS7.

The PKCS7 function operates by padding an octet-aligned message to a multiple of B bytes, where B is typically a cipher's blocksize. 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

[16] We also mounted 900GB of ephemeral EC2 storage to each instance as a temporary filesystem to save CNF files during operation.

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 Appendix F. 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 $G_0 \leftarrow \mathsf{F}_{\mathsf{PKCS7}}$.[17] This evaluation was performed on AWS, E5, and E7.

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 a full resulting attack appears in Appendix E. 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 1 shows one attack transcript at $t = 5, \delta = 0.5$. Additional experiments measuring the effect of t on this format are provided in Appendix E. 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. At a high level, if someone wants to use a CRC to protect their data from non-malicious errors in transit, they will treat the data stream they wish to compute over as a polynomial and then take its remainder with respect to the CRC generator polynomial and then attach this value at the end of the stream. A receiver can then recompute the CRC over the beginning of the data stream and check for equality. If the verification check fails, then an error has occurred in transmission. CRCs are well known to be malleable, due to the linearity of the functions: namely, for a CRC it is always the case that $\text{CRC}(a \oplus b) = \text{CRC}(a) \oplus \text{CRC}(b)$. 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.[18] A key feature of this format is that a valid ciphertext C^* should *not* be vulnerable to a format oracle attack using a

[17] In practice, this plaintext distribution tends to produce messages with short padding.

[18] In our implementation we used a simple implementation that does not reflect input and output, or add an initial constant value before or after the remainder is calculated.

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 E5.

5.3 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 Trew 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. Additional CRC experiments can be found in Appendix E. 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

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

where the CRC of $d_{15}\ldots d_0$ 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: $r(x) \oplus (\sum_{i=8}^{15} d'_i * x_i) \pmod{b(x)}$. If our guess for the plaintext was correct, the new CRC is correct for the truncated message, since $\mathrm{CRC}(d_{15}\ldots d_0) \oplus \mathrm{CRC}(d'_{15}...d'_8\|0\ldots 0\}) = \mathrm{CRC}(0\ldots 0\|d_7\ldots d_0)$. 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 [arm18], 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,[19] in an average of ˜13.3 queries, with each full attack taking only seconds on E5.

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 [arm18], 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 [AWS15]. 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:

$$[RANDOMSECRET][TIMEFIELD][CIPHERSUITE][PROTOCOLVERSION][FORMATVERSION]$$

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 occurence and has been seen in the wild in at least one internet wide scan [BZD$^+$16]. To evaluate a realistic attack on a practical format function, we developed a format checking function for the Amazon s2n [AWS15] TLS session ticket format. s2n uses 60-byte tickets with a 12-byte header comprising a protocol version, ciphersuite version, and format version, along with an 8-byte timestamp that is compared against the current server clock. Although s2n uses authenticated encryption (AES-GCM), we consider a hypothetical scenario where nonce re-use has allowed for message forgery [Fer05, BZD$^+$16].

[19] 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.

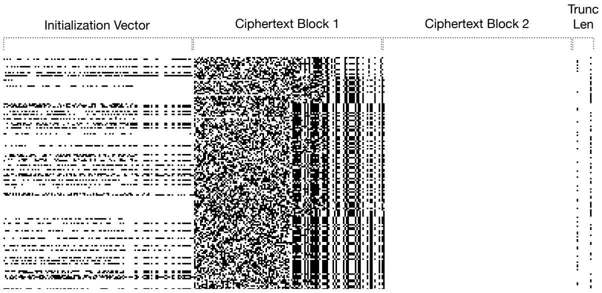


Figure 4: A contiguous set of malleation queries made by `Delphinium` during a simulated CBC attack. The rightmost bits signal truncation (from left or right).

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.

5.4 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.[20] 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 blocksize 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 4 gives a brief snapshot of this pattern of malleations discovered by `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.,* [RHF+17, Ran18, ppm]. 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 iff the puzzle is valid.

[20] In practice, truncation in CBC simply removes blocks from either end of the ciphertext.

Because existing implementations *e.g.,* [ppm] 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 Appendix E.5.

6 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 [MRLG15, BFK$^+$12, PDM$^+$18, RD10, YPM05, PY04, JS11, AF18, KMSS15, GGK$^+$16], 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], [PBP$^+$17] focuses on generating public input values that lead to maximum leakage of secret input in Java programs where leakage is defined in terms of channel capacity and shannon entropy. Unlike our work, Pasareanu et al. [CSP16] do not consider an adversary that makes *adaptive* queries based on results of previous oracle replies. Both [CSP16] and [PBP$^+$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, ZCSE16]. However, many other techniques have been proposed in the literature. Several works propose sophisticated multi-query approach with high accuracy [SM19, CMV16], resulting in the `ApproxMC` tool we use in our experiments. Other works examine the complexity of parity constraints [ZCSE16], and optimize the number of variables that must be constrained over to find a satisfying assignment [IMMV15].

7 Conclusion

Our work leaves a number of open problems. In particular, we proposed several optimizations that we were not able to implement in our tool, due to time and performance constraints. 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.,* [GLMP18]. 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.

Acknowledgments

The authors would like to thank Amazon Web Services for partially funding our evaluation through EC2 credits, Mate Soos and Kuldeep Meel for answering relevant questions on ApproxMC and Probably Approximate Model Counters, Rolf Rolles for general help with SAT and SMT solver libraries, and finally Dr. Nadia Heninger and her students for granting us access to and helping us use their compute cluster.

References

[ABD$^+$15] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella-Béguelin, and Paul Zimmermann. Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 5–17, New York, NY, USA, 2015. ACM.

[AF18] Gildas Avoine and Loïc Ferreira. Attacking GlobalPlatform SCP02-compliant Smart Cards Using a Padding Oracle Attack. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(2):149–170, May 2018.

[AKMY10] Abdel Alim Kamal and Amr M. Youssef. Applications of SAT Solvers to AES key Recovery from Decayed Key Schedule Images. *IACR Cryptology ePrint Archive*, 2010:324, 07 2010.

[AP13] Nadhem J. AlFardan and Kenneth G. Paterson. Lucky Thirteen: Breaking the TLS and DTLS Record Protocols. In *IEEE S&P (Oakland) '13*, pages 526–540, 2013.

[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, 2015. `https://eprint.iacr.org/2015/1129`.

[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, pages 16–26, Washington, DC, USA, 2009. IEEE Computer Society.

[arm18] ARM Architecture Reference Manual, Mar 2018.

[ASS$^+$16] Nimrod Aviram, Sebastian Schinzel, Juraj Somorovsky, Nadia Heninger, Maik Dankel, Jens Steube, Luke Valenta, David Adrian, J. Alex Halderman, Viktor Dukhovni, Emilia Käsper, Shaanan Cohney, Susanne Engels, Christof Paar, and Yuval Shavitt. DROWN: Breaking TLS using SSLv2. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 689–706, Austin, TX, 2016. USENIX Association.

[AWS15] Introducing s2n, a New Open Source TLS Implementation. `https://aws.amazon.com/blogs/security/introducing-s2n-a-new-open-source-tls-implementation/`, June 2015.

[BBDL$^+$15] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P. Strub, and J. K. Zinzindohoue. A Messy State of the Union: Taming the Composite State Machines of TLS. In *2015 IEEE Symposium on Security and Privacy*, pages 535–552, May 2015.

[BDP03] Fahiem Bacchus, Shannon Dalmao, and Toniann Pitassi. DPLL with Caching: A new algorithm for #SAT and Bayesian inference. *Electronic Colloquium on Computational Complexity (ECCC)*, 10, 01 2003.

[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*, volume 6, pages 21–21, Berkeley, CA, USA, 1996. USENIX Association.

[BFK$^+$12] Romain Bardou, Riccardo Focardi, Yusuke Kawamoto, Lorenzo Simionato, Graham Steel, and Joe-Kai Tsay. Efficient Padding Oracle Attacks on Cryptographic Hardware. In *CRYPTO '12*, volume 7417 of LNCS, pages 608–625. Springer, 2012.

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

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

[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. *ACM Trans. Inf. Syst. Secur.*, 7(2):206–241, May 2004.

[BL99] Elazar Birnbaum and Eliezer L. Lozinskii. The Good Old Davis-Putnam Procedure Helps Counting Models. *J. Artif. Int. Res.*, 10(1):457–477, June 1999.

[BN00] Mihir Bellare and Chanathip Namprempre. Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm. In Tatsuaki Okamoto, editor, *ASIACRYPT 2000*, pages 531–545, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[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)*, pages 307–322, April 2018.

[BZD$^+$16] Hanno Böck, Aaron Zauner, Sean Devlin, Juraj Somorovsky, and Philipp Jovanovic. Nonce-Disrespecting Adversaries: Practical Forgery Attacks on GCM in TLS. In *10th USENIX Workshop on Offensive Technologies (WOOT 16)*, Austin, TX, 2016. USENIX Association.

[CDM17] Dmitry Chistikov, Rayna Dimitrova, and Rupak Majumdar. Approximate counting in SMT and value estimation for probabilistic programs. *Acta Informatica*, 54(8):729–764, Dec 2017.

[CFM$^+$14] Supratik Chakraborty, Daniel J Fremont, Kuldeep S Meel, Sanjit A Seshia, and Moshe Y Vardi. Distribution-aware sampling and weighted model counting for SAT. In *Twenty-Eighth AAAI Conference on Artificial Intelligence*, 2014.

[CFMV15] Supratik Chakraborty, Dror Fried, Kuldeep S Meel, and Moshe Y Vardi. From weighted to unweighted model counting. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

[CHVV03] Brice Canvel, Alain Hiltgen, Serge Vaudenay, and Martin Vuagnoux. Password interception in a SSL/TLS channel. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003*, pages 583–599, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[CMV16] 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)*, 7 2016.

[COQ09] Nicolas T. Courtois, Sean O'Neil, and Jean-Jacques Quisquater. Practical Algebraic Attacks on the Hitag2 Stream Cipher. In Pierangela Samarati, Moti Yung, Fabio Martinelli, and Claudio A. Ardagna, editors, *Information Security*, pages 167–176, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[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)*. IEEE, June 2016.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A Machine Program for Theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

[dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[DP60] Martin Davis and Hilary Putnam. A Computing Procedure for Quantification Theory. *J. ACM*, 7(3):201–215, July 1960.

[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, pages 493–504, New York, NY, USA, 2010. ACM.

[Ecl20] Eclypsium. Perilous Peripherals: The Hidden Dangers Inside Windows & Linux Computers, Feb 2020.

[EGSS14] Stefano Ermon, Carla P. Gomes, Ashish Sabharwal, and Bart Selman. Low-Density Parity Constraints for Hashing-Based Discrete Integration, 2014.

[Fer05] Niels Ferguson. Authentication Weaknesses in GCM, 05 2005.

[FHK$^+$14] Martin Franz, Andreas Holzer, Stefan Katzenbeisser, Christian Schallhart, and Helmut Veith. CBMC-GC: an ANSI C compiler for secure two-party computations. In *International Conference on Compiler Construction*, pages 244–249. Springer, 2014.

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

[GGK$^+$16] Christina Garman, Matthew Green, Gabriel Kaptchuk, Ian Miers, and Michael Rushanan. Dancing on the Lip of the Volcano: Chosen Ciphertext Attacks on Apple iMessage. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 655–672, Austin, TX, 2016. USENIX Association.

[GLMP18] Paul Grubbs, Marie-Sarah Lacharite, Brice Minaud, and Kenneth G. Paterson. 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, pages 315–331, New York, NY, USA, 2018. ACM.

[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, pages 54–61. AAAI Press, 2006.

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

[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/`, September 2010.

[IMMV15] Alexander Ivrii, Sharad Malik, Kuldeep Meel, and Moshe Vardi. On computing minimal independent support and its applications to sampling and counting. *Constraints*, 21, 08 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`, January 2006.

[JS11] Tibor Jager and Juraj Somorovsky. How to Break XML Encryption. In *ACM CCS '2011*. ACM Press, October 2011.

[Kal98] Burt Kaliski. PKCS #7: Cryptographic Message Syntax Version 1.5. RFC 2315, March 1998.

[KMSS15] Dennis Kupser, Christian Mainka, Jörg Schwenk, and Juraj Somorovsky. How to Break XML Encryption – Automatically. In *Proceedings of the 9th USENIX Conference on Offensive Technologies*, WOOT'15, Berkeley, CA, USA, 2015. USENIX Association.

[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`, September 2014.

[MGC$^+$16] Benjamin Mood, Debayan Gupta, Henry Carter, Kevin Butler, and Patrick Traynor. Frigate: A validated, extensible, and efficient compiler and interpreter for secure computation. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 112–127. IEEE, 2016.

[MIM$^+$19] Jens Müller, Fabian Ising, Vladislav Mladenov, Christian Mainka, Sebastian Schinzel, and Jörg Schwenk. PDF Insecurity. Available at `https://www.pdf-insecurity.org/encryption/encryption.html`, November 2019.

[min] MiniSAT. Available at `http://minisat.se/`.

[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*, pages 244–258, 2005.

[MRLG15] Florian Maury, Jean-Rene Reinhard, Olivier Levillain, and Henri Gilbert. Format Oracles on OpenPGP. In *Topics in Cryptology - CT-RSA 2015, The Cryptographer's Track at the RSA Conference 2015*, pages 220–236, San Francisco, United States, April 2015.

[MSA$^+$19] Robert Merget, Juraj Somorovsky, Nimrod Aviram, Craig Young, Janis Fliegenschmidt, Jörg Schwenk, and Yuval Shavitt. Scalable Scanning and Automatic Classification of TLS Padding Oracle Vulnerabilities. In *Proceedings of the 28th USENIX Conference on Security Symposium*, SEC'19, pages 1029–1046, Berkeley, CA, USA, 2019. USENIX Association.

[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, IETF Secretariat, January 2020. `http://www.ietf.org/internet-drafts/draft-ietf-suit-information-model-05.txt`.

[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*, pages 127–143, Berlin, Heidelberg, 2016. Springer-Verlag.

[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)*, volume 4121 of *Lecture Notes in Computer Science*, pages 102–115. Springer, August 2006.

[NPWB18] Aina Niemetz, Mathias Preiner, Clifford Wolf, and Armin Biere. Btor2 , btormc and boolector 3.0. In Hana Chockler and Georg Weissenbacher, editors, *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*, volume 10981 of *Lecture Notes in Computer Science*, pages 587–595. Springer, 2018.

[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, pages 427–437, New York, NY, USA, 1990. ACM.

[PBP$^+$17] Quoc-Sang Phan, Lucas Bang, Corina S. Pasareanu, Pasquale Malacaria, and Tevfik Bultan. Synthesis of Adaptive Side-Channel Attacks. In *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*. IEEE, August 2017.

[PDM$^+$18] Damian Poddebniak, Christian Dresen, Jens Müller, Fabian Ising, Sebastian Schinzel, Simon Friedberger, Juraj Somorovsky, and Jörg Schwenk. Efail: Breaking S/MIME and OpenPGP Email Encryption using Exfiltration Channels. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 549–566, Baltimore, MD, 2018. USENIX Association.

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

[png] PNG (Portable Network Graphics) Specification, Version 1.2.

[ppm] ppmx. Z3 Sudoku Solver. Available at `https://github.com/ppmx/sudoku-solver`.

[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.

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

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

[RHF$^+$17] Heinz Riener, Finn Haedicke, Stefan Frehse, Mathias Soeken, Daniel Große, Rolf Drechsler, and Goerschwin Fey. metaSMT: focus on your application and not on solver integration. *International Journal on Software Tools for Technology Transfer*, 19(5):605–621, 2017.

[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 Serge Vaudenay, editor, *EUROCRYPT 2006*, pages 373–390, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[SBB$^+$04] Tian Sang, Fahiem Bacchus, Paul Beam, Henry Kautz, and Toniann Pitassi. Combining Component Caching and Clause Learning for Effective Model Counting. In *SAT 2004*, 05 2004.

[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)*, 1 2019.

[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`, May 2012.

[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)*, pages 244–257, 06 2009.

[SRS+18] Shiqi Shen, Soundarya Ramesh, Shweta Shinde, Abhik Roychoudhury, and Prateek Saxena. Neuro-Symbolic Execution: The Feasibility of an Inductive Approach to Symbolic Execution, 2018.

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

[stp] The Simple Theorem Prover (STP). Available at `https://stp.github.io/`.

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

[Tod91] Seinosuke Toda. PP is As Hard As the Polynomial-time Hierarchy. *SIAM J. Comput.*, 20(5):865–877, October 1991.

[Val79] Leslie G. Valiant. The Complexity of Enumeration and Reliability Problems. *SIAM J. Comput.*, 8(3):410–421, 1979.

[Vau02] Serge Vaudenay. Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS. In *EUROCRYPT '02*, volume 2332 of LNCS, pages 534–546, London, UK, 2002. Springer-Verlag.

[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, pages 1313–1328, New York, NY, USA, 2017. ACM.

[VV86] L.G. Valiant and V.V. Vazirani. NP is as easy as detecting unique solutions. *Theoretical Computer Science*, 47:85 – 93, 1986.

[W3C17] W3C. Web Cryptography API. Available at `https://www.w3.org/TR/WebCryptoAPI/`, January 2017.

[WBL+19] Shuai Wang, Yuyan Bao, Xiao Liu, Pei Wang, Danfeng Zhang, and Dinghao Wu. Identifying Cache-Based Side Channels through Secret-Augmented Abstract Interpretation. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 657–674, Santa Clara, CA, August 2019. USENIX Association.

[WS05] Wei Wei and Bart Selman. A New Approach to Model Counting. In Fahiem Bacchus and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing*, pages 324–339, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[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*, pages 299–319, 2005.

[Zal] Michal Zalewski. Technical "whitepaper" for afl-fuzz.

[ZCSE16] Shengjia Zhao, Sorathan Chaturapruek, Ashish Sabharwal, and Stefano Ermon. Closing the Gap Between Short and Long XORs for Model Counting. In *AAAI 2016*, 12 2016.

A Proof that DeriveAttack is Greedy

A.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. $\mathsf{Max\#SAT}(X,Y,G)$ 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 $\mathsf{\#SAT}((M_0,M_1),G_{i-1}) - \mathsf{\#SAT}((M_0,M_1),G_i)$. The new constraint depends upon the O_{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) [CFMV15]. The message length is known to the attack algorithm.

Let O_{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 conjuction of the following constraints:

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

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

$$\begin{aligned} &\mathsf{F}(\mathsf{Maul}_{\mathsf{plain}}(M_0, \mathbf{S}_i)) = \\ &\mathsf{F}(\mathsf{Maul}_{\mathsf{plain}}(M_1, \mathbf{S}_i)) = \\ &\ O_{\mathsf{dec}}(\mathsf{Maul}_{\mathsf{ciph}}(C, \mathbf{S}_i)) \end{aligned}$$

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

A.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#SAT aborts will be considered.

A.2.1 Base Case

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

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

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.

$\mathsf{Max\#SAT}(S, M_0 || M_1, G_0)$ 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 $p = \frac{min(\mathsf{\#SAT}(M_0, G_0), \mathsf{\#SAT}(M_1, G_0))}{x}$. 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 O_{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 $\mathsf{Max\#SAT}$. 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.

A.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.

A.2.3 Max#SAT Aborts

Max#SAT may abort. This implies no satisfying assignment for X or Y can be found, implying the formula is inconsistent, or it implies 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.

B 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 B.1 *For the format* F *and malleation* $\mathsf{Maul}_{\mathsf{plain}}$ *pair* $(\mathsf{F}_{\mathsf{Parity}}, \mathsf{Truncation})$*, the greedy attack is optimal.*

B.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.[21]

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.[22] 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.

B.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.

[21] Here we will consider the parity of strings of less than 1 bit in length to be undefined.

[22] 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.

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.

B.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.

B.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.

B.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.

C The Greedy Algorithm can be Suboptimal

In the following proof sketch, we provide a counterexample to the optimality of the greedy algorithm as both 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.[23]

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

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 C.1 *For the format* F *and malleation* $\mathsf{Maul}_{\mathsf{plain}}$ *pair* $(\mathsf{F}_{\vee}, Truncation \circ Exclusive\text{-}OR)$*, the greedy attack is not optimal.*

[23] The attack algorithm then can flip any subset of bits, and truncate the message down to any non-zero length.

C.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 $2^k > l$.[24]

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, $\mathsf{F}_\vee$ considers the leftmost $k+1$ bits, comparing them to some per-instance randomly sampled “cookie” value[25]. 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:

C.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.

C.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.

[24] We will consider l chunks of size k when analyzing this distribution.

[25] 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 disparate from reality.

C.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.

C.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[26] 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.

$$\begin{aligned} \text{exp. profitability} &\approx 2^{L-(k+2)} \\ \text{exp. queries} &= 2^{L-(k+1)} + L - (k+2) \end{aligned}$$

C.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.

[26] Any non-zero amount of truncation leaving at least one non-cookie bit is equally effective but complicates the attack slightly.

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.

C.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 [png, Pfe03].

C.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.

D Realizing Plaintext Weight Functions

As discussed in §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 probability.

This approach can produce counter-intuitive results in practice. One example manifests in our attacks on the PKCS7 encryption padding scheme [Kal98], which we discuss in §5. 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 PKCS7 padding lengths are equally likely. A naive weighting function adds a constraint for each possible message which returns a weight. However, such an approach suffers from exponential runtime overhead. To efficiently 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. Notice also that the original attack algorithm is equivalent to using a trivial weighting function where $\forall M \in \mathcal{M}$ it holds that $W(M) = B$.

E Additional Experimental Results

E.1 PKCS7

Figure 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 PKCS7 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.

E.2 Bitwise Padding Format Tests Over *t*

We empirically observe the parameter δ has limited impact on solver runtime and set $\delta = 0.5$.[27] 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 6 demonstrates malleation string bitmaps for full attacks against the bitwise padding format for $t \in \{1,\ldots,5\}$. 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. Figures 7 and 8 demonstrate `ApproxMC` evaluations of counts of removed messages across t with logarithmically- and full-sized parity constraints, respectively.

To evaluate the impact of changing t on our attacks against the bytewise PKCS7 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 the `ApproxMC` model counting tool to measure and average the minimum profitability of the resulting malleation string.[28] 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. The output data of these microbenchmarks is omitted for brevity.

[27] i.e. requiring all trials to be satisfied.

[28] We did this by simulating each possible result from the decryption oracle, and using `ApproxMC` the number of plaintext candidates eliminated by each.

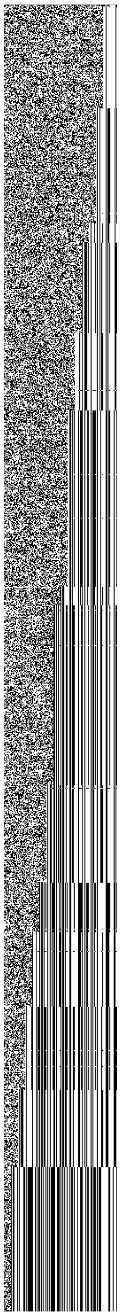


Figure 5: PKCS7 malleation bitmap

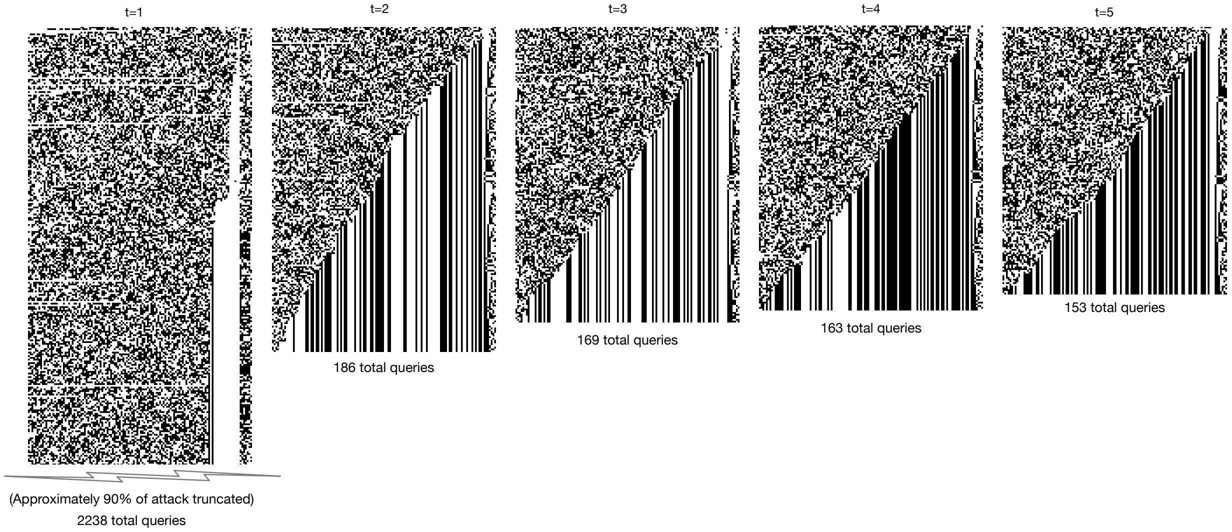


Figure 6: Malleation strings for 128-bit bitwise padding ($\mathsf{F}_{\mathsf{bitpad}}$) using a stream cipher. Experiments consider several different values of t, all with $\delta = 0.5$. Each uses a different random 128-bit message, which can account for some variability in the attack progress. For $t = 1$ the total number of queries is too large to show, and so we only present approximately the first 10% of the full attack transcript.

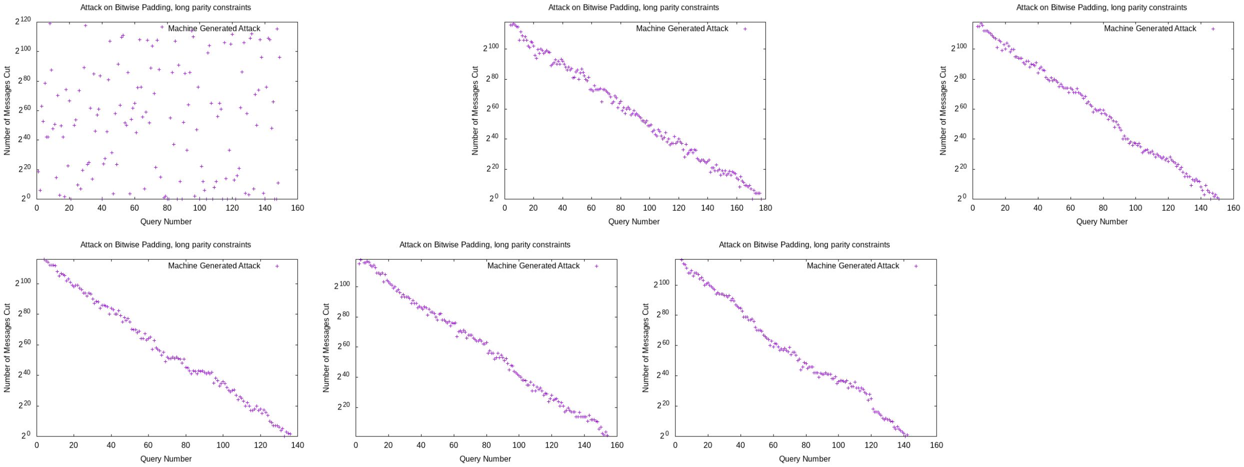


Figure 7: Approximations of how many messages were removed in a 128-bit Bitwise Padding Format attack with long parity constraints.

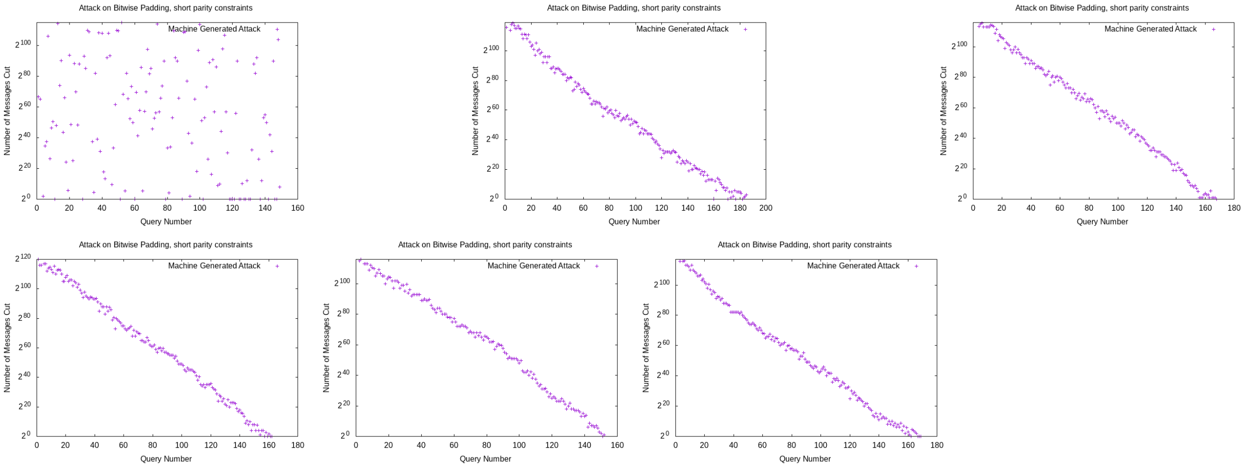
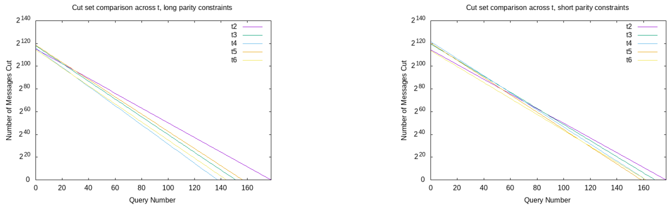


Figure 8: Approximations of how many messages were removed in a 128-bit Bitwise Padding Format attack with short parity constraints.

E.3 Bitwise Padding Format Test: Differences between Short and Long exclusive-OR



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

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

Figure 9: Attacks on 128-bit Bitwise Padding Format.

E.4 CRC

Figure 10 displays a bitmap of malleation strings for an attack against CRC, allowing truncation and exclusive-OR malformations.

E.5 Sudoku

In Figure 11 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 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.

Figure 10: 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.

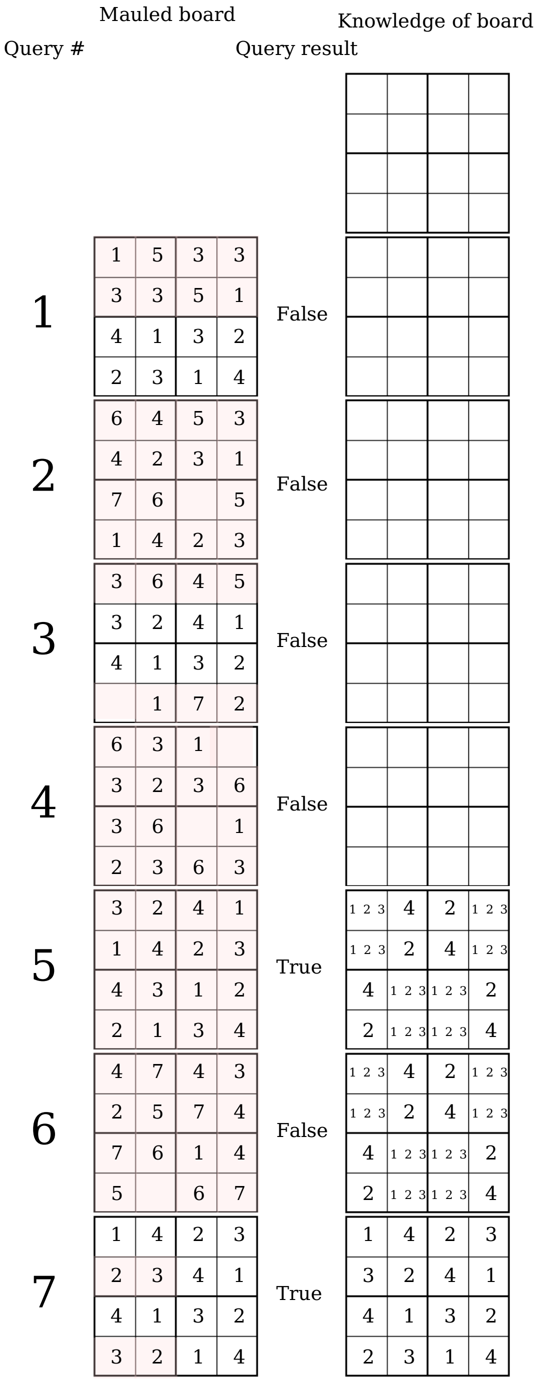
Other attacks took up to 33 queries, but have a similar pattern (deriving notable information from *True* oracle results) and thus are omitted for brevity.

F Format Check Implementations

We present a listing of several Python-Solver format check functions below.

F.1 PKCS7 padding $\mathsf{F}_{\mathsf{PKCS7}}$

```
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
```



Query #	Mauled board				Query result	Knowledge of board			
1	1	5	3	3	False				
	3	3	5	1					
	4	1	3	2					
	2	3	1	4					
2	6	4	5	3	False				
	4	2	3	1					
	7	6		5					
	1	4	2	3					
3	3	6	4	5	False				
	3	2	4	1					
	4	1	3	2					
		1	7	2					
4	6	3	1		False				
	3	2	3	6					
	3	6		1					
	2	3	6	3					
5	3	2	4	1	True	1 2 3	4	2	1 2 3
	1	4	2	3		1 2 3	2	4	1 2 3
	4	3	1	2		4	1 2 3	1 2 3	2
	2	1	3	4		2	1 2 3	1 2 3	4
6	4	7	4	3	False	1 2 3	4	2	1 2 3
	2	5	7	4		1 2 3	2	4	1 2 3
	7	6	1	4		4	1 2 3	1 2 3	2
	5		6	7		2	1 2 3	1 2 3	4
7	1	4	2	3	True	1	4	2	3
	2	3	4	1		3	2	4	1
	4	1	3	2		4	1	3	2
	3	2	1	4		2	3	1	4

Figure 11: 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.

```
            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
```

F.2 AWS Session Ticket $\mathsf{F}_{\mathsf{Ticket}}$

```
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
```

F.3 Bitwise padding $\mathsf{F}_{\mathsf{bitpad}}$

```
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)
```