Using SMT Solvers to Automate 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 this progress, 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 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 first gained practical salience with the discovery of *padding oracle* attacks on a number of standard encryption protocols [Vau02, Hun10, BFK+12, AP13, MDK14]. Despite repeated warnings to industry, variants of these attacks continue to plague modern systems, including TLS 1.2's CBC-mode ciphersuite [AP13, AP15] 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 likely remain undetected, due to the difficulty of exploiting non-standard format oracles.

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 exploits in OpenPGP clients [GGK+16, PDM+18] were closed only when cryptographers 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 novel 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?*

The key requirement of our approach is that the software responsible for developing this attack should require no further assistance from human beings. Moreover, we require that the developed attack 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 encryption protocols. While our techniques are designed to be general, we stipulate that 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 a wide variety of such functions.

Relationship to previous automated attack work. Previous work [BRB18, QSP17, 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 (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[1]. The cited works perform symbolic execution to (adaptively) discover chosen inputs that will reveal as much information as possible about the secret input. Like our techniques, several of these works employ SAT solvers and model counting techniques. However, beyond these superficial similarities, there are fundamental differences from our approach: (1) in our techniques, we use the solver to explore the *full* format checking function, rather than analyzing a program instruction-by-instruction, (2) perhaps because of this approach, our results operate over much larger secret domains than the cited works. To illustrate the differences, our experimental results consider hidden message spaces of size 2^{128} and beyond, with malleation strings of equivalent sizes, while the cited works appear limited to much smaller secret sizes that rarely even reach 2^{24}. Moreover, our format functions are relatively complex. It is an open question to determine whether the more limited experimental results in the cited works due to fundamental limitations of the symbolic execution approach, or if the two approaches can be combined in some fruitful way.

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 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`) as an open source tool for use and further development by the research community[2].
- We validate our tool experimentally, deriving several attacks using different format-checking functions, including padding schemes such as PKCS #7, session ticket checking functions, and contrived functions such as a Sudoku verifier. These experiments represent, to our knowledge, the first evidence of a completely functioning end-to-end machine-developed format oracle attack.

1.1 Overview of our Techniques

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

[1] To see this, imagine that we encrypt the secret input as a ciphertext. Let each malleation string produced by our attack represent some "low" input. Now define a malleation function that appends the low input to the encrypted secret; a decryption function $\mathsf{Decrypt}$ that decrypts the secret input and passes both the decrypted secret and the (unmodified) "low" input to a format checking function; and a format checking function F that evaluates the two inputs and produces a result in $\{0, 1\}$. An equivalent transformation may exist in the opposite direction.

[2] We provide an *anonymized* source code repository containing our implementation at: `https://github.com/Pythia3431/Delphinium`.

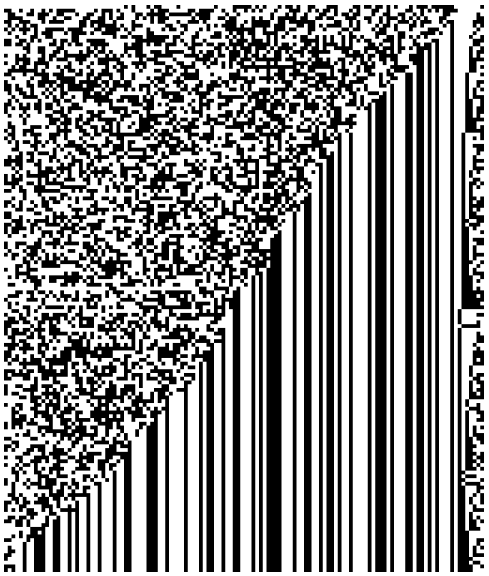


Figure 1: Output of a format oracle attack that our algorithms developed against a bitwise padding check oracle $\mathsf{F}_{\mathsf{bitpad}}$ (see §5.3 for a full description). The original ciphertext is a valid 128-bit (random) padded message encrypted using a stream cipher. Each row of the bitmap represents a *malleation string* that was exclusive-ORed with the ciphertext prior to making a decryption query.

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 certain symmetric encryption schemes. Because encryption schemes can be quite complex, we do not wish to reason over the scheme itself. Instead, for a given encryption scheme Π, we require the user to develop two efficiently-computable functions. The function $\mathsf{Maul}^{\Pi}_{\mathsf{ciph}}(C, S) \rightarrow C'$ takes as input a valid ciphertext and some opaque *malleation instruction string* S (henceforth "malleation string"), and produces a new, mauled ciphertext C'. The function $\mathsf{Maul}^{\Pi}_{\mathsf{plain}}(M, S) \rightarrow M'$ computes the equivalent malleation over some plaintext, producing a plaintext (or, in some cases, a set of possible plaintexts[3]). 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 basic stream cipher can be realized by defining both functions as bitwise exclusive-OR. However, malleation functions may also implement features such as truncation or complex editing, guided by instructions drawn from S.

Building block: theory solvers. As a key building block in our techniques, we will make use of efficient theory solvers, such as SAT and Satisfiability Modulo Theories (SMT) [stp, Mic]. 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. For simplicity of exposition, the remainder of this section will elide the specific details of

[3]In §2 we will formalize and extend this definition to include schemes with *non-unique* (or key-dependent) malleation effects, such as CBC-mode encryption.

these solvers: we will simply assume a "black-box solver" that can find a satisfying assignment using 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 our setting, each experiment comprises a specific malleation string S that can be applied to the target ciphertext C^* using the ciphertext malleation function. The technical challenge is therefore the need to programmatically derive a malleation string that (a) represents a meaningful input to the malleation functions, and (b) ensures that the attack will make *progress*, *i.e.,* eliminate some messages at each iteration.

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 may 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 $G_0 : \{0,1\}^n \rightarrow \{0,1\}$ that expresses all known constraints over the value of M^*[4]. (If we do not have any *a priori* knowledge about the distribution of M^*, we can set this initial formula G_0 to specify that M^* is any plaintext[5]).

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 a concrete assignment for S that induces specific properties on M_0, M_1. Specifically, we require that each

[4]Here n represents an upper bound on the length of the plaintexts.

[5]If the encryption scheme does not have a specific format for plaintexts, G_0 can be an empty constraint formula. Alternatively, in the case where it is known that M^* satisfies the format check, the attacker can set $G_0 \leftarrow \mathsf{F}$.

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 for the three abstract bitvectors (M_0, M_1, S) that satisfies the following constraint formula:

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

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 $\bar{S}_i$, and proceed to the next step.

Step 2: Query the oracle; update the constraints. Given a concrete malleation string $\bar{S}_i$, we now apply the ciphertext malleation function to compute an experiment ciphertext $C \leftarrow \mathsf{Maul}_{\mathsf{ciph}}(C^*, \bar{S}_i)$, and submit C to the decryption oracle. When the oracle produces a result $b \in \{0,1\}$, we compute an updated constraint formula G_i as follows:

$$G_i(X) \leftarrow (G_{i-1}(X) \ \wedge \ \mathsf{F}(\mathsf{Maul}_{\mathsf{plain}}(X, \bar{S}_i)) = b)$$

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

Step 3: Attack completion. The attack concludes when the solver is unable to identify a satisfying assignment in Step (1). In the ideal case, this occurs because the constraint system G_{i-1} admits only one possible candidate plaintext: if this occurs, we can employ the solver to directly recover M^* from G_{i-1}. 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 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 determine).

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 a fundamental implication of the constraint system we construct: our approach forces the solver to ensure that each malleation string S implicitly partitions the cadidate 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.

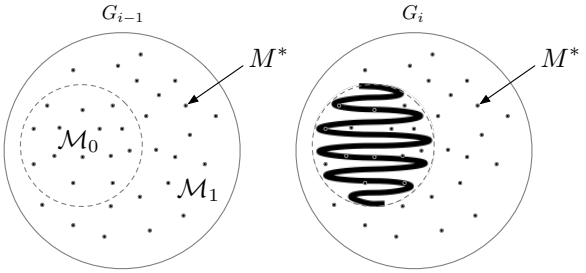


Figure 2: Left: illustration of a plaintext candidate space defined by G_{i-1}, highlighting the two subsets $\mathcal{M}_0, \mathcal{M}_1$ induced by a specific malleation string $\bar{S}_i$. Right: the candidate space defined by G_i, in which many candidates have been eliminated following an oracle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

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 the experiment is executed and the constraint formula is 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 $\bar{S}_i$ that defines two disjoint subsets $\mathcal{M}_0, \mathcal{M}_1$ of the current candidate plaintext space, such that for $b \in \{0, 1\}$ and $\forall M \in \mathcal{M}_b$ it holds that $\mathsf{F}(\mathsf{Maul}_{\mathsf{plain}}(M, \bar{S}_i)) = b$. In this description, any concrete decryption oracle result must "rule out" (at a minimum) every plaintext contained in the subset $\mathcal{M}_{\hat{b}-1}$ from the candidate plaintext space. It is easy to see that $\psi(G_{i-1}, G_i)$ is equal to the cardinality of $\mathcal{M}_{\hat{b}-1}$.

To increase the profitability of a given query, it is therefore necessary to maximize the size of $\mathcal{M}_{\hat{b}-1}$. Of course, since we do not know the value $\hat{b}$ prior to issuing a decryption oracle query, the obvious strategy is to find $\bar{S}_i$ such that *both* $\mathcal{M}_0, \mathcal{M}_1$ are as large as possible. (Put slightly differently, we wish to find an experiment $\bar{S}_i$ 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 [Mic], 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)[6].

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 finely distributed throughout the space, a condition that is likely in our setting.

Approximating the count. While exact model counting seems intractable for our application, a number of works have explored *approximate* solutions to the problem [GSS06, CMV16, SM19]. One powerful

[6]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.

technique, initially proposed by Valiant and Vazirani [VV86], based on work by Stockmeyer [Sto83] and applied practically by Gomes *et al.* [GSS06], uses a SAT oracle as follows: given a constraint formula F over some bitvector T, add to F a series of s random parity constraints, each computed over the bits of T. For $j = 1$ to s, the j^{th} parity constraint can be viewed as requiring that $H_j(T) = 1$ where $H_j : \{0,1\}^{|T|} \rightarrow \{0,1\}$ is a universal hash function. Intuitively, each additional constraint reduces the number of satisfying assignments approximately by half, independently of the underlying distribution of valid solutions. The implication is as follows: if a satisfying 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. This approach can further be formulated as an optimization problem [FRS17].

To apply this technique to our attack, we extend the algorithm 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[7]. For a sufficiently high value of s, this approach effectively eliminates many "unprofitable" malleation string candidates and thus significantly improves the efficiency of the attack.

The main weakness of this approach stems from the probabilistic nature of the approximation algorithm. Even when 2^s satisfying assignments exist for M_0, M_1, the solver may deem the extended formula unsatisfiable with relatively high probability. In our approach, this false-negative will cause the algorithm to reduce the size of s, potentially resulting in the selection of a less-profitable experiment S. Following Gomes *et al.* [GSS06], we are able to substantially improve our certainty by conducting multiple *trials* t within each query. For example, rather than conducting the above check one time and rejecting s if the solver fails to find a satisfying solution, we instead conduct t separate instances of the check (each using new, randomized parity constraints) and accept 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 Section 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,

[7]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.

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. We will define $\mathsf{Truncate}_m(M)$ as a truncation function that outputs the first m bits of the bitstring M.

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, and the deterministic algorithm $\mathsf{Decrypt}_K(C)$ decrypts C to produce a plaintext or the distinguished error symbol $\perp$. We require that the encryption scheme is *correct*, in the sense that for all valid M, K it holds that $\mathsf{Decrypt}_K(\mathsf{Encrypt}_K(M)) = M$. 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. We now formalize this description.

Definition 2 (Malleation functions) The *malleation functions* for a symmetric encryption scheme Π comprise a pair of efficiently-computable functions $(\mathsf{Maul}^\Pi_{\mathsf{ciph}}, \mathsf{Maul}^\Pi_{\mathsf{plain}})$ with the following properties. Let $\mathcal{M}, \mathcal{C}$ be the plaintext (resp. ciphertext) space of Π. The function $\mathsf{Maul}^\Pi_{\mathsf{ciph}} : \mathcal{C} \times \{0,1\}^* \to \mathcal{C} \cup \{\perp\}$ takes as input a ciphertext and a *malleation instruction string*. It outputs a ciphertext or the distinguished error symbol $\perp$. The function $\mathsf{Maul}^\Pi_{\mathsf{plain}} : \mathcal{M} \times \{0,1\}^* \to \hat{\mathcal{M}}$, on input a plaintext and a malleation instruction string, outputs a *set* $\hat{\mathcal{M}} \subseteq \mathcal{M} \cup \{\perp\}$ of plaintexts, possibly including the distinguished 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 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 $\mathsf{Maul}^{\Pi}_{\mathsf{ciph}}(C, S) \neq \perp$:

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

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 plaintexts* (and possibly the decryption error symbol $\perp$). This captures the possibility that certain forms of malleation can induce unpredictable key-dependent effects when ciphertexts are decrypted[8]. Finally, we note that a description of these functions is necessary, but not *sufficient* for our attacks to work. Malleation functions can "overfit" an encryption scheme; for example, one can define a trivial $\mathsf{Maul}^{\Pi}_{\mathsf{plain}}$ such that for every possible input, $\mathsf{Maul}^{\Pi}_{\mathsf{plain}}$ simply outputs the set $\mathcal{M} \cup \{\perp\}$. Similarly, a plaintext malleation function can be formulated even for schemes that are explicitly *non*-malleable, such as authenticated encryption modes. The effectiveness of our attacks depends on both the malleability features of the scheme, and the precision with which the malleation functions describe them.

2.1.1 Example Malleation Functions

We now briefly describe several concrete encryption schemes and their associated malleation functions.

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 length truncation, in which bits of ciphertext (resp. plaintext) are removed from the right side. 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 $r\|S'$, where $r \geq 0$ is decoded as an integer indicating the desired length of the output string. The function(s) output $\mathsf{Truncate}_{|C|-r}(C) \oplus S'$ (resp. $\mathsf{Truncate}_{|C|-r}(M) \oplus S'$).

CBC and CFB mode. In CBC mode, any bits altered in ciphertext block i will display the same pattern of changes in block $i+1$ following decryption, while block i will be replaced with pseudorandom output (excepting $i = 0$). CFB is similar, except that changes to block i will emerge in block i, while block $i+1$ will be replaced. To describe this scheme, the functions $\mathsf{Maul}^{\Pi_{\mathsf{CBC}}}_{\mathsf{plain}}$ and $\mathsf{Maul}^{\Pi_{\mathsf{CFB}}}_{\mathsf{plain}}$ each output a *set* of plaintexts comprising every possible candidate[9]. In §4 we describe a more efficient encoding of this output.

2.2 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 $\mathsf{F} : \mathcal{M} \cup \{\perp\} \to \{0,1\}$ as a *format checking function.* Our

[8] 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.

[9] This set will contain a separate plaintext candidate for each possible string of size 2^ℓ in every such block.

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 (which we will discuss in more detail below). 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 [Mic] 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.

2.3 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[10]. For the purposes of describing our algorithms, we specify a query to the solver by the subroutine $\mathsf{Solve}\{(A_1, \ldots, A_N) : G\}$ where $A_1, \ldots, A_N$ 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 $(a_1, \ldots, a_n)$, (2) the distinguished response unsat, or (3) the error $\mathsf{unknown}$, which indicates that the solver could not identify or rule out a satisfying assignment.

Model counting. While SAT determines the existence of a single satisfying assignment, a more general variant of the problem, #SAT, determines the *number* of satisfying assignments for a given set of constraints[11]. In the literature this problem is known as *model counting* [Val79, GSS06, BL99, BJP00] [SBB+04, WS05, BDP03, CFMV15].

Very early work on this problem utilizes the David Putnam algorithm for finding satisfying assignments. Modifications included seeing how many free variables remain after a satisfying assignment was found only fixing values for a subset of variables, recursively finding the number of solutions in connected components dynamically, and then caching results of these solutions to subproblems so that a SAT solver would not need to solve the same problem twice. [SBB+04]. Later on, more sophisticated techniques were proposed with better efficiency on complex formulas. These techniques attempted to limit the number of queries to a SAT oracle to be only logarithmic and they

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

[11] 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 #SAT.

utilized uniform sampling of the solution space to do this, whether that be through random walks or hashing constraints [GSS06, WS05]. As discussed in §1.1, our approach relies on approximate model counting techniques in order to optimize attack progress.

The approach we use is an adaptation of the `MBound` algorithm developed by Gomes *et al.* [GSS06]. This algorithm checks for the presence of 2^s satisfying assignments for a bitvector T, by adding to the formula a collection of s randomly-generated parity constraints: $P_1(T) = 1 \ \wedge \ \ldots \ \wedge \ P_s(T) = 1$. To improve performance, Gomes *et al.* propose to perform up to t independent trials (each using different randomized parity constraints), where at least $\lceil (\frac{1}{2} + \delta)t \rceil$ trials must be successful. We employ the parameters t, δ as adjustable inputs to our algorithms.

From counting to an optimization metric. Our algorithms reformulate model counting as an *optimization* problem. This problem was previously considered by Fremont *et al.* [FRS17] under the term Max#SAT. In this setting, our goal is to identify an assignment to one variable (or set of variables) that maximizes the number of satisfying assignments to a second variable. In our setting, we wish to identify a malleation string S that induces an (approximately) optimal partitioning of the plaintext space. Our algorithms do this by employing the model counting techniques described above: repeatedly querying the solver at different set sizes until we find a malleation string that maximizes the size of both sets.

This formulation leaves an open theoretical question. Recall that the solver is now solving for some string S that induces a specific relation between a separate bitvector (M_0 or M_1) and the parity constraints applied to it. This fundamentally changes the statistical argument that underpins the model counting schemes from which we derive our inspiration. Rather than rely on these arguments, we instead verified the effectiveness of the technique empirically.

More prosaically, this optimization technique induces a fundamental tradeoff between computational efficiency and query profitability. In previous work on model counting [SM19, GSS06] improved accuracy of a model count requires that the solver conduct multiple independent trials. Unlike the counting techniques of [GSS06], these trials cannot be broken across multiple independent solver queries. Instead, each trial must be conducted as part of a single solver query, since the same abstract value S must be referenced by each trial. This increases the complexity of the formula fed to the solver, and requires that we carefully balance the runtime of each attack against the query profitability by adjusting the parameters t, δ.

3 Our Constructions

In this section we present a high-level description of our main contribution: a complete algorithm for programmatically conducting a format oracle attack. We first present pseudocode for the basic algorithm itself. We then describe a number of optimizations and improvements that improve performance and handle special cases.

3.1 The Attack Algorithm

Algorithms 1, 2 present the core algorithms of our main attack generation procedure. The entry routine of our attack is DeriveAttack, given in Figure 1. This routine takes as input a target ciphertext C^*, constraint formulae for the functions $\mathsf{Maul}_{\mathsf{plain}}, \mathsf{F}$, as well as a *solver-interpretable* implementation of the function $\mathsf{Maul}_{\mathsf{ciph}}$. It additionally takes in the maximum plaintext and malleation string lengths n, m as well as the model counting parameters t, δ described in the previous section. The

algorithm repeatedly queries the decryption oracle $\mathcal{O}_{\mathsf{dec}}$, and terminates by outputting a (possibly incomplete) description of $M^* \in \{0,1,*\}^n$, where the wildcard $*$ indicates an unknown bit.

Adjusting the conjectured set size. Our algorithms employ an abstract subroutine `AdjustSize` that is responsible for updating the conjectured set size s in our optimization loop. We define this routine to use the following pseudocode API:

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

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'. By maximizing the conjectured set sizes, the attack elicits a malleation string at each iteration which maximizes the number of messages eliminated in expectation. As such, this algorithm is greedy, optimizing given the constraints at each iteration to discover locally optimal malleation strings. Appendices A and B outline cases where this greedy approach is and is not globally optimal, respectively. We discuss implementation strategies for the `AdjustSize` routine in §4.2.

Additional subroutines. The subroutine `FindKnownBits` uses the solver to find a unique solution for M^* given a constraint formula. If a unique solution cannot be found, it instead decides which bit positions of M^* can be uniquely determined, and outputs a string of the form $\{0,1,*\}^n$. The subroutine `ParityConstraint` constructs a randomized parity constraint of weight k over a bitvector of size n. We omit these basic procedures for brevity.

3.2 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 E we discuss a more nuanced approach that allows the user to *weight* plaintexts by their probability of occurring in M^*.

4 Prototype Implementation

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

Algorithm 1: `DeriveAttack`

Input: Description of $(\mathsf{F}, \mathsf{Maul}_{\mathsf{plain}})$; code for $\mathsf{Maul}_{\mathsf{ciph}}$; target ciphertext C^*; max plaintext and malleation string length n, m; initial constraints G_0 over $\{0,1\}^n$; $t \in \mathbb{N}$; $\delta \in [0.0, 0.5]$

Output: $M^* \in \{0, 1, *\}^n$

Procedure:

$Z \leftarrow \bot$;
$i \leftarrow 1,\ b_{\mathsf{success}} \leftarrow \textsc{True},\ s \leftarrow n$;
while $b_{\mathsf{success}} = \textsc{True}$ **do**
  $b_{\mathsf{continue}} \leftarrow \textsc{True}$;
  while b_{continue} **do**
    $(b_{\mathsf{success}}, \bar{S}_i) \leftarrow$ `FindMalleation`$(G_{i-1}, s, t, \delta, n, m,$ $\mathsf{F}, \mathsf{Maul}_{\mathsf{plain}})$;
    $(b_{\mathsf{continue}}, s, Z) \leftarrow$ `AdjustSize`$(b_{\mathsf{success}}, n, s, Z)$;
  if $b_{\mathsf{success}} = \textsc{True}$ **then**
    `// Query the decryption oracle`
    $b \leftarrow \mathcal{O}_{\mathsf{dec}}(\mathsf{Maul}_{\mathsf{ciph}}(C^*, \bar{S}_i))$;
    $G_i(X) \leftarrow \big[G_{i-1}(X) \wedge (\mathsf{F}(\mathsf{Maul}_{\mathsf{plain}}(X, \bar{S}_i)) = b)\big]$;
  $i \leftarrow i + 1$;
return `FindKnownBits`(G_{i-1});

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. It outputs the (partial) recovered plaintext.

SMT/SAT solver. Our implementation supports multiple SMT solver frameworks (STP [stp] and Z3 [Mic]) 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 $\mathsf{Maul}^{\Pi}_{\mathsf{ciph}}$. In practice, this module is provided as a Python file, although it can execute code written in another language such as C.

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.

Algorithm 2: `FindMalleation`

Input: Constraint formula G; $s \in \mathbb{W}$, $t \in \mathbb{N}$, $\delta \in [0.0, 0.5]$; max plaintext and malleation string length n, m; F; $\mathsf{Maul}_{\mathsf{plain}}$

Output: $b_{\mathsf{success}} \in \{0,1\}$; $\bar{S}_i \in \{0,1\}^m$

Procedure:

```
// Construct parity constraints
```

$k \leftarrow \lceil log_2(n) \rceil$;

for $l \leftarrow 1$ **to** t **do**

  for $m \leftarrow 1$ **to** $2s$ **do**

    $H_{l,m} \leftarrow$ `ParityConstraint` (n, k)

```
// Query the solver
```

$(\mathsf{result}, \bar{S}_i) \leftarrow \mathsf{Solve}\{(M_{1,0}, M_{1,1}, \ldots, M_{t,0}, M_{t,1}, S) :$
$[\forall j \in [1,t], b \in \{0,1\} :$
$(G(M_{j,0}) = G(M_{j,1}) = 1 \ \wedge$
$\mathsf{F}(\mathsf{Maul}_{\mathsf{plain}}(M_{j,b}, S)) = b)] \wedge$
$\exists \mathcal{S}_1 \subseteq [1,t], |\mathcal{S}_1| \geq \lceil (0.5+\delta)t \rceil, \forall j \in \mathcal{S}_1 :$
$(\forall k \in [1,s] : (H_{j,k}(M_0) = 1)) \wedge$
$\exists \mathcal{S}_2 \subseteq [1,t], |\mathcal{S}_2| \geq \lceil (0.5+\delta)t \rceil, \forall j \in \mathcal{S}_2 :$
$(\forall k \in [1,s] : H_{j,s+k}(M_1) = 1)\}$;

if $\mathsf{result} = \mathsf{sat}$ **then**

  return (TRUE, $\bar{S}_i$);

else

  return (FALSE, $\perp$);

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

The discussion so far has allowed us to ignore many implementation and performance tradeoffs. However, realizing our algorithms in a practical tool required us to solve a number of challenging engineering problems. Many stem from the performance and capability limitations of existing SAT and SMT solvers.

Selecting SAT and SMT solvers.. During the course of this work we evaluated several SMT and SAT solvers, including Z3 [Mic], Boolector[12] [BNPB], STP [stp] and CryptoMiniSAT [SNC09]. We determined that each package offers significant performance and functionality tradeoffs. Our final design uses Z3 as a front-end SMT solver, due to its robust feature set and strong performance in simplifying complex constraint formulae. Unfortunately, while Z3 has many advantages, it has one critical *disadvantage*: the default SAT solver (MiniSAT [min]) does not perform Gaussian elimination of XOR 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.

[12] Unfortunately we found Boolector (via the Python bindings) unusable due to software issues.

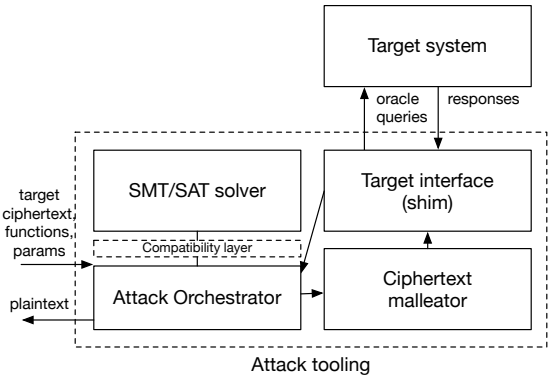


Figure 3: Architecture of `Delphinium`.

Combining Z3 and CMS via a CNF bridge. To improve performance, we adopted CryptoMiniSAT (henceforth CMS), which has been used extensively in the model counting literature [SNC09] due to its ability to simplify XOR 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 constraint processing 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. The main limitation in this approach is that Z3's CNF generation tactic is time consuming and single-threaded. Thus we do not generate fresh CNF with each query: instead, we employ Z3 to periodically generate a new, simplified CNF formula. This approach represents a tradeoff that obtains many of the benefits of Z3's simplification tactics, while also avoiding a bottleneck on CNF generation. It is, however, still deeply suboptimal from a performance perspective, and we note that there is a great deal of room for performance improvement[13]. Nonetheless, we found that it systematically outperformed all other alternatives.

Low-density parity constraints. Our implementation of model counting requires our tool to incorporate $2st$ distinct parity functions into each solver query. Each full-weight parity constraint comprises an average of $\frac{n}{2}$ XOR gates (where n is the maximum length of M^*), resulting in a complexity increase of tens to hundreds of gates in our experiments. To address this, we adopted an approach used by several previous model counting 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

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

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[14].

Describing $\mathsf{Maul}_{\mathsf{plain}}$ and 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. We discuss malleation functions further in Appendix D.

4.3 Test Harness

For our stream cipher experiments in Section 5 we developed a test harness to implement the Ciphertext Malleator and Target Interface shim. This test harness simulates the effect of mauling and decrypting M^* using a given malleation string $\bar{S}_i$. At the start of the attack, this harness samples a random plaintext M^* from an appropriate distribution. To respond to decryption queries it directly calculates $\mathsf{F}(\mathsf{Maul}_{\mathsf{plain}}(M^*, \bar{S}_i))$.

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.

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. Finally, in pursuing this prototype, we submitted various patches to the underlying theory solvers. These patches have since been included in the upstream software projects.

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

[14] 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 XOR expansion.

[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) 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.

Finally, 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[15]. 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 $\bar{S}_i$ produced by our attack, which allows us to "replay" any transcript after the fact. To measure the algorithm's performance in eliminating plaintext candidates, we applied the `ApproxMC` approximate model counting tool [SM19] to several of these replayed attack transcripts: this tool provides us with an estimate for the total number of remaining candidates for M^* at every phase of a given attack.

5.2 Parameter Tuning

As a precursor to conducting complete attacks, we conducted a number of experiments to optimize our software, and to identify practically effective values for the adjustable parameters in our algorithms.

Selecting attack parameters. The main adjustable parameters in our counting algorithms are t, the number of counting trials, and δ, which determines the fraction that must succeed. The parameter δ has limited impact on solver runtime; we experientially found that 0.5[16] produces

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

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

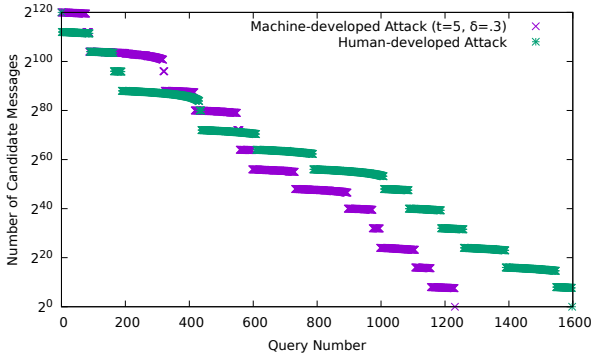


Figure 4: Number of remaining plaintext candidates satisfying G_i (y-axis) vs. decryption oracle query number i (x-axis), counted using `ApproxMC`. Two attack runs are depicted on PKCS #7 with a 128-bit stream cipher. The machine-developed attack uses $t = 5, \delta = 0.5$. Our settings for `ApproxMC` permit a counting error factor of $\pm.8x$ with a certainty of 0.8. These error bars are difficult to see on a log scale, so we omit them for clarity.

reasonable performance (this is consistent with previous works [GSS06]). We determined that the key parameter affecting attack runtime is t. Empirical 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 solver runtime.

To evaluate the impact of changing t on our attacks, we conducted a series of "microbenchmarks" of our attack's experiment generation procedure, using the PKCS #7 format checking function (described in the following section) as a concrete input. In each experiment, we asked the algorithm to derive 10 malleation strings $\bar{S}_i$ at some phase of a complete attack run, and then used the `ApproxMC` model counting tool to measure and average the minimum profitability of the resulting malleation string[17]. We repeated each experiment for each $t \in \{1, \ldots, 6\}$, and measured the wall-clock execution time of experiment generation. Our results indicate that $t = 5$ provides an effective tradeoff between query efficiency and execution time, with higher values of t producing diminishing returns. We use this value to conduct our full attacks.

5.3 End-To-End Evaluation

PKCS7 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 this function as a benchmark both because it is reasonably complex, and because the human-developed attack is well understood.

PKCS #7 operates by padding an octet-aligned message to a multiple of B bytes. The padding string is of length $1 \leq P \leq B$ bytes, and each byte comprises an 8-bit encoding of the integer P. The format checking function $\mathsf{F}_{\mathsf{PKCS7}}$ recovers P from the final byte of the padded message, verifies that $P \in \{1, \ldots, B\}$, and checks that each of the trailing P bytes contains the same value. We present a Z3-Python implementation in Appendix F.

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

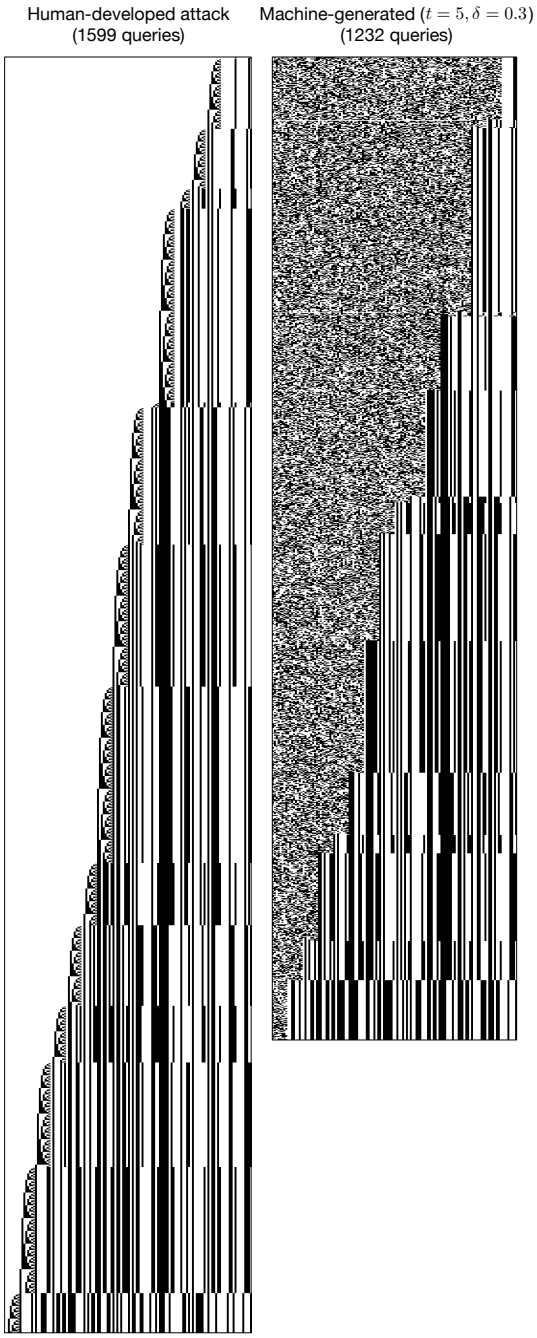


Figure 5: Malleation strings produced by attacks on PKCS #7 (128-bit) using a stream cipher. The i^{th} row of each bitmap depicts $\bar{S}_i$, with "1" bits in black (note that each diagram has been scaled to fill the column width, so pixels are not square). The left rectangle presents the output of a *human-developed* attack completing in 1599 queries. The right shows a machine-developed attack at $(t = 5, \delta = 0.5)$ completing in 1232 queries.

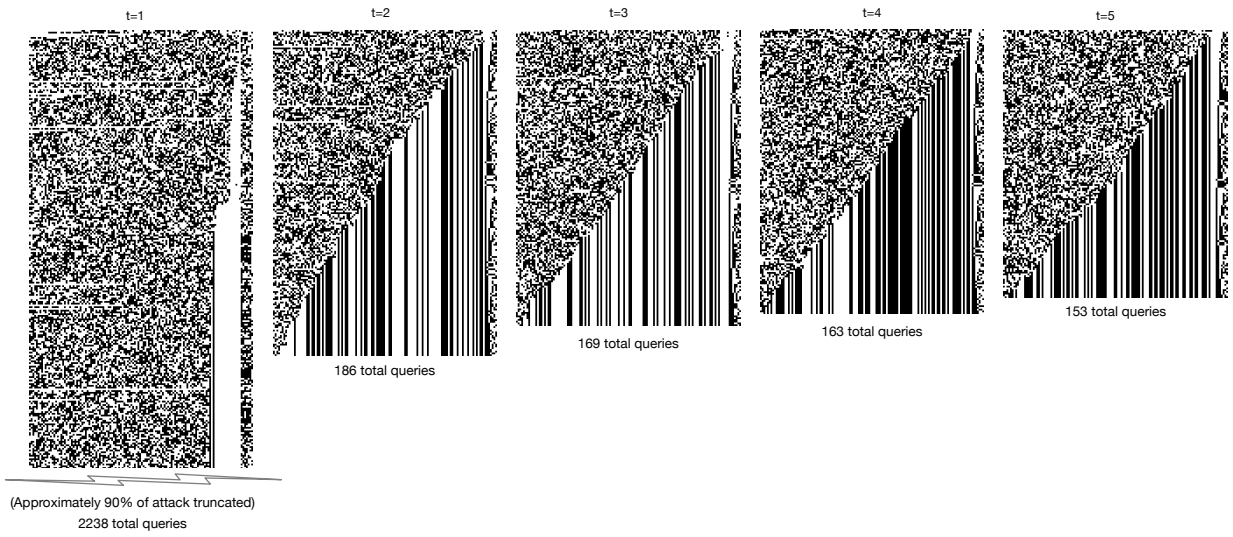


Figure 6: Malleation strings for 128-bit bitwise padding ($\mathsf{F}_{\mathsf{bitpad}}$) using a stream cipher. Experiments consider several different values of t, all with $\delta = 0.5$. Each uses a different random 128-bit message, which can account for some variability in the attack progress. For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

Experimental 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 $G_0 \leftarrow \mathsf{F}_{\mathsf{PKCS7}}$[18]. Due to the time requirements of the full attack, we ran one complete attack on E5, although we ran many partial attacks and attacks at lower parameters. As a baseline comparison, we also implemented the manually-developed "human" attack (based on Vaudenay's approach [Vau02]).

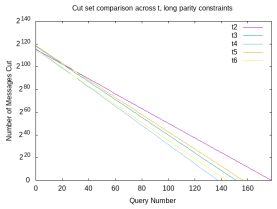
Results. Our attack run completed in 1232 queries over 12.26 hours, averaging 35.82 seconds per oracle query. While this runtime appears significantly better than the ~2000 query expectation we calculated for the human attack (and even the 1599 queries in the human attack example we present), the reader should not draw any conclusions from this single data point. A detailed analysis of query profitability (see Figure 4) shows that our attack has approximately the same typical profitability as the human attack, and query count differences can be attributed to normal variability. This test was run on E5.

Figure 5 presents a transcript of the malleation strings produced by our attack algorithm, along with the malleations devised by the human attack. We optimized our attack to re-generate fresh (simplified) CNF whenever the decryption oracle produced a TRUE result: this CNF generation consumes much of the runtime. Appendix 2 shows the time required to generate CNF, as well as the time CMS required in order to extract each query $\bar{S}_i$. To measure the query profitability of our attack, we employed the `ApproxMC` model counting tool to measure the number of remaining candidates for M^* at each phase of the attack. This result, along with the conjectured set size s is shown in Figure 4.

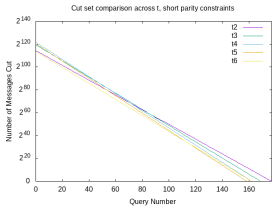
Bitwise Padding. In the course of testing our attacks, we constructed a simplified *bit* padding scheme $\mathsf{F}_{\mathsf{bitpad}}$. This contrived scheme encodes the bit length of the padding P into the rightmost

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

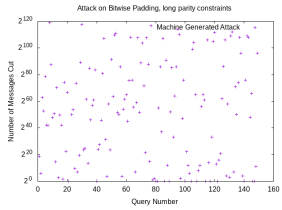
$\lceil log_2(n) \rceil$ bits of the plaintext string, and then places up to P padding bits directly to the left of this length field, with each padding bit set to 1. A useful property of this padding format for our experiments is that it possesses a high theoretical query profitability: a single decryption query can obtain approximately one bit of the underlying message. We verified the effectiveness of our attacks against this format, using a 128-bit stream cipher at $t = 5$, $\delta = 0.5$ on $\mathsf{E5}$, obtaining a complete attack run in 151 queries with an average query time of 28.88 seconds. Figure 1 shows one attack transcript at $t = 5, \delta = 0.5$, and Figure 6 shows multiple random attack runs at values of $t \in \{1, \ldots, 5\}$.



(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 $\log n$.



(c) Approximate number of messages taken out for each query when parameter $t = 1$, with long parity constraints.

Figure 7: Tool attack on 128-bit Bitwise Padding Format.

In order to further evaluate our attack algorithm, as well as to compare behavior over attack parameters and to validate the logarithmically-sized hash functions described in Section 4.2, we conducted a series of attacks against this bitwise padding format similar to those we provide for PKCS7. 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, however, unlike in the PKCS7 experiment, we model count 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.

Figure 7b demonstrates the `ApproxMC` approximations of messages ruled out by each query using short hash functions, and Figure 7a demonstrates them for full hash functions. Figure 7c shows results for $t = 1$, truncating the attack which required over 2000 queries[19]. Trend line averages are provided in the referenced figures; full scatter plot graphs are provided in Appendix C.3.

From these experiments we observe diminishing returns beyond $t = 2$ for the bitwise format, and similar behavior between short and long hash functions.

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.

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

[19]The $t = 1$ graphs are similar for the two hash function classes.

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 C.4.

S2N Session Ticket Format. Although our experiments are incomplete at the time of submission, we additionally developed a format checking function against 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 authenticator forgery [Fer05, BZD+16].

We implemented the `s2n` ticket format verification routine as a function $\mathsf{F}_{\mathsf{ticket}}$ (using a best guess for the server's current time value at each query) and executed it against a truncated ticket of length 16 bytes. Our attack is running successfully (and eliminating plaintext candidates) at submission time; we will update this section when it completes.

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], [QSP17] has focused on generating public input values that lead to maximum leakage of secret input in Java programs. The ultimate goal of these tools is to learn the sequence of values that result in the most information gain of the secret via channel capacity and shannon entropy measurements. While [CSP16] does not consider an adversary that makes *adaptive* queries (i.e. those that are a result of information previously learned) [QSP17] does account for this in their algorithm constructions. However, to reiterate, these works focus on java programs and utilize symbolic execution to get all different paths that a program can take and they are specifically interested in timing and memory usage side channels. While there analysis of shannon entropy utilizes model counting methods, these methods are based on numerical optimization methods and the solving of non linear maximal optimization problems. Their results are valid for smaller input spaces with complex programs, but appear from the experimental section to be untested on larger domain spaces, such as those with greater than 2^{100} elements. We instead focus on a slightly more specific problem of tasks done with encrypted data and the insecurities resulting from performing computation with these values after decryption. Sequential works follow this model, utilizing symbolic execution and numeric optimization methods to recover secret input [BRB18].

Using solvers for cryptographic tasks. 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]. A separate line of works uses solvers to generate cryptographic protocols [JMKG14, HKM15, AGH13, AGH15].

Approximate model counting. In this work we use model counting techniques pioneered by Gomes *et al.* [GSS06]. 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 [IMMV15].

7 Conclusion

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

Open questions and future work. 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, and 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.

[AGH13] Joseph A Akinyele, Matthew Green, and Susan Hohenberger. Using SMT solvers to automate design tasks for encryption and signature schemes. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 399–410. ACM, 2013.

[AGH15] Joseph A. Akinyele, Christina Garman, and Susan Hohenberger. Automating fast and secure translations from Type-I to Type-III pairing schemes. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1370–1381, New York, NY, USA, 2015. ACM.

[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`.

[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.

[BNPB] Armin Biere, Aina Niemetz, Mathias Preiner, and Robert Brummayer. Boolector. Available at `https://boolector.github.io/`.

[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.

[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.

[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.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, July 1960.

[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.

[HKM15] Viet Tung Hoang, Jonathan Katz, and Alex J. Malozemoff. Automated analysis and synthesis of authenticated encryption schemes. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 84–95, New York, NY, USA, 2015. ACM.

[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.

[JMKG14] Alex J. Malozemoff, Jonathan Katz, and M.D. Green. Automated analysis and synthesis of block-cipher modes of operation. *Proceedings of the Computer Security Foundations Workshop*, 2014:140–152, 11 2014.

[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.

[Mic] Microsoft Research. The Z3 theorem prover. Available at `https://github.com/Z3Prover/z3`.

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

[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.

[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.

[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.

[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.

[QSP17] Corina S. Pasareanu Pasquale Malacaria Tevfik Bultan Quoc-Sang Phan, Lucas Bang. Synthesis of adaptive side-channel attacks. In *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*. IEEE, August 2017.

[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. 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/`.

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

A.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.[20]

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.[21] 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.

A.2 Observations

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

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

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

Truncating additional bits in the first query has a similar effect. Consider truncating two bits in the first query: rather than two cases, there are four (the four values of two truncated bits): two cases which leave the un-truncated bits valid (unchanged parity), and two which flip parity. A partition is formed over messages ending in 01 or 10 and messages ending in 00 or 11, and these two

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

[21] 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.

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.

A.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.

A.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.

A.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.

B 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[22].

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

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

B.1 Preliminaries

Consider the message distribution of uniform L-bit strings. The size of this message space is then 2^L. Let[23] $kl = L$ and we require that $2^k > l$.

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

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

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

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

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

When checking a message, $\mathsf{F}_\vee$ considers the leftmost $k+1$ bits, comparing them to some per-instance randomly sampled "cookie" value[24]. 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:

B.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.

B.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.

B.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

[24] 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.

such queries from oracle results on random strings, and thus these queries yield trivial information about the target plaintext.

B.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[25] 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.

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

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

B.4 The Greedy Attack

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

The greedy attack will follow this attack path because doing so differentiates a class of messages which match the guess of the last k bits from the remaining which have some other last k bits. As

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

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.

B.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].

B.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

Table 1: Generated CNF file size for implemented format check functions
Note that number of trials linearly increases CNF size, $t = 5$.

Format	**File size**
Sudoku	1931051 (1.9M)
128-bit Bitwise Padding	4436905 (4.3M)
128-bit PKCS #7	606882 (593K)
TLS s2n Ticket	870549 (851K)[26]

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.

C Additional Experimental Results

C.1 Generated CNF for Format Checks

In Figure 1, generated CNF formula file sizes are provided. CNF complexity, and thus wall-clock execution time in solvers, trends with file size, although other factors such as interconnectedness of boolean operators (akin to Boolean Circuit depth) determine this overhead.

C.2 PKCS #7

Figure 2 shows wall-clock time in seconds to generate CNF formulae (using Z3), and to extract malleation strings from the resulting CNF formulae (using CMS) is recorded over the course of a PKCS #7 attack. Note that CNF re-generation only occurs periodically. We optimize the attack to perform such re-generation only occurs when additional bits of the candidate message model are uniquely constrained. Between each CNF re-generation, the time required to extract malleation strings increases linearly by a factor dependent on t and the complexity of the format check.

[26] Varies due to format check being stateful, s2n ticket validation involves steps such as a timestamp check

Query	**Known bits**	**CNF time** (sec)	$\bar{S}_i$ **time** (sec)
0	0	22.17	18.70
79	16	219.75	10.34
88	24	253.72	9.68
320	32	814.51	9.78
324	40	849.16	8.56
418	48	1102.34	7.91
551	56	1363.24	8.13
559	64	1300.01	5.88
598	72	1433.81	2.79
732	80	1695.01	3.57
897	88	1921.50	3.45
975	96	2096.14	3.96
998	104	2061.03	3.82
1108	112	2143.65	3.91
1157	120	2222.60	1.59

Table 2: Time required to generate experiments in a 128-bit PKCS #7 attack run. Known bits indicates the number of bits of * that have been uncovered; CNF time indicates the number of seconds required to extract a CNF formula for the SAT solver; $\bar{S}_i$ time indicates the number of seconds required to extract a malleation string. Note that query extractions run in parallel as described in Section 4.

.

C.3 Bitwise Padding Format Tests Over t

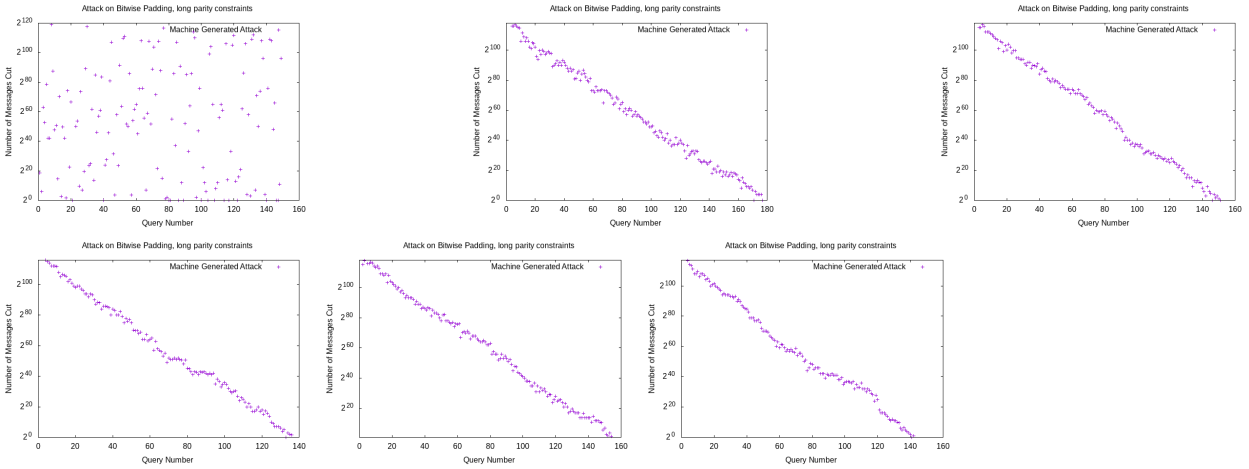


Figure 8: Approximations of how many messages were removed in a 128-bit Bitwise Padding Format attack with long parity constraints.

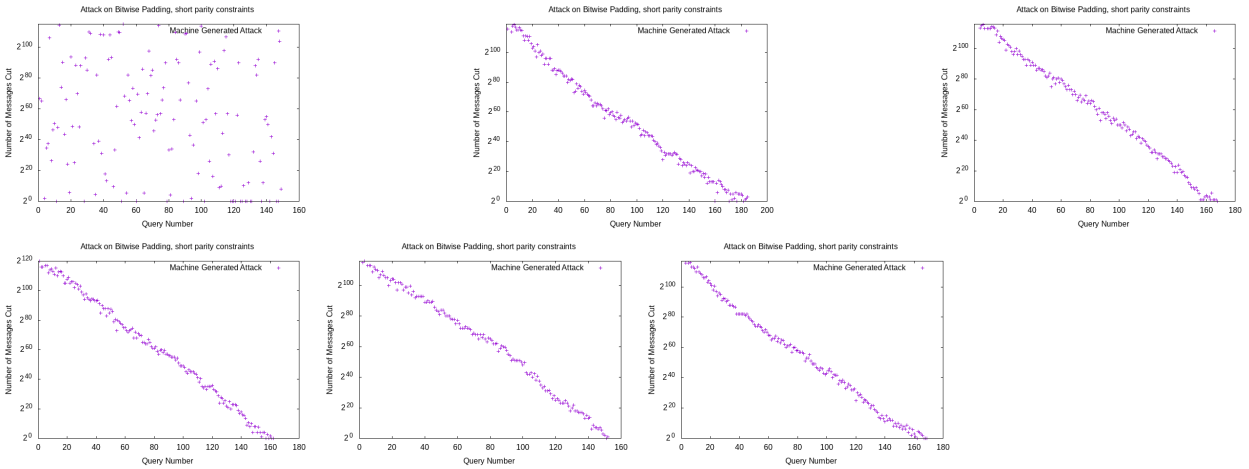


Figure 9: Approximations of how many messages were removed in a 128-bit Bitwise Padding Format attack with short parity constraints.

C.4 Sudoku

In Figure 10 we show an attack against the Sudoku format check. This format maps a 48-bit bitvector to entries into a 4x4 Sudoku board and checks the validity of the resulting solution. Values are coded as 3 bits, allowing entries from 0 to 7 (where 0 is coded as an unfilled square). Initially, the solver knows nothing about the real plaintext, which is a given Sudoku solution. The mauled boards show the malleation chosen by the solver applied to the real plaintext. After 7 queries, the solver has uniquely constrained its model for the real plaintext, which is correct. When the solver chooses a malleation which causes the resulting Sudoku board to remain valid, it can derive many additonal constraints on the values of its model for the real plaintext. Other attacks took up to 33

queries, but have a similar pattern (deriving notable information from $True$ oracle resutls) and thus are omitted for brevity.

D Implementing Malleation Functions

In this section we describe plaintext malleation functions for several common encryption schemes. The simplest of these functions, $\mathsf{Maul}^{\Pi_{\mathsf{stream}}}_{\mathsf{plain}}$, can be realized using the bitvector XOR function provided by Z3. However, support for the remaining functions requires some specific changes to our implementation.

Supporting plaintext truncation. Support for plaintext truncation requires that we support plaintexts of variable length, something that is not natively provided by the abstract bitvector interface of the solver. This requires that we encode a notion of plaintext length into each bitvector that represents a plaintext, and make corresponding changes to the interface for F. For plaintexts of length $\ell \leq n$ we accomplish this simply by treating the first $log_2(n)$ bits of each bitvector as a *length field*, and specifying that every implementation of F decode this value prior to evaluating the plaintext.

CBC mode. Our general formulation of CBC malleation specifies that $\mathsf{Maul}^{\Pi_{\mathsf{CBC}}}_{\mathsf{plain}}$ outputs a *set* of plaintexts. Supporting this representation efficiently requires significant changes throughout our system. First, we modify the interface of $\mathsf{Maul}^{\Pi_{\mathsf{CBC}}}_{\mathsf{plain}}$ to output two abstract bitvectors (M, Mask). This second bitvector represents a *masking string*: any bit position j where $\mathsf{Mask}[j] = 1$ is viewed as a *wildcard* in the message vector M: *i.e.,* 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 formulation requires that we modify F to take (M, Mask) as input. The function must then reason over both M and Mask. Finally, we modify the output of F to produce three possible output values: $0, 1, 2$. The new output value (2) indicates that the format check cannot assign a definite true/false value on this input, due to the uncertainty created by the unconstrained bits.[27] Realizing this formulation requires only minor implementation changes to our core algorithms.

E 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 "quality".

This approach can produce counterintuitive results in practice. One example manifests in our attacks on the PKCS #7 encryption padding scheme [Kal98], which we discuss in §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

[27] In practice, we implement the output of F as a bitvector of length 2, and modify our algorithms to use 00, 01, in place of 0, 1.

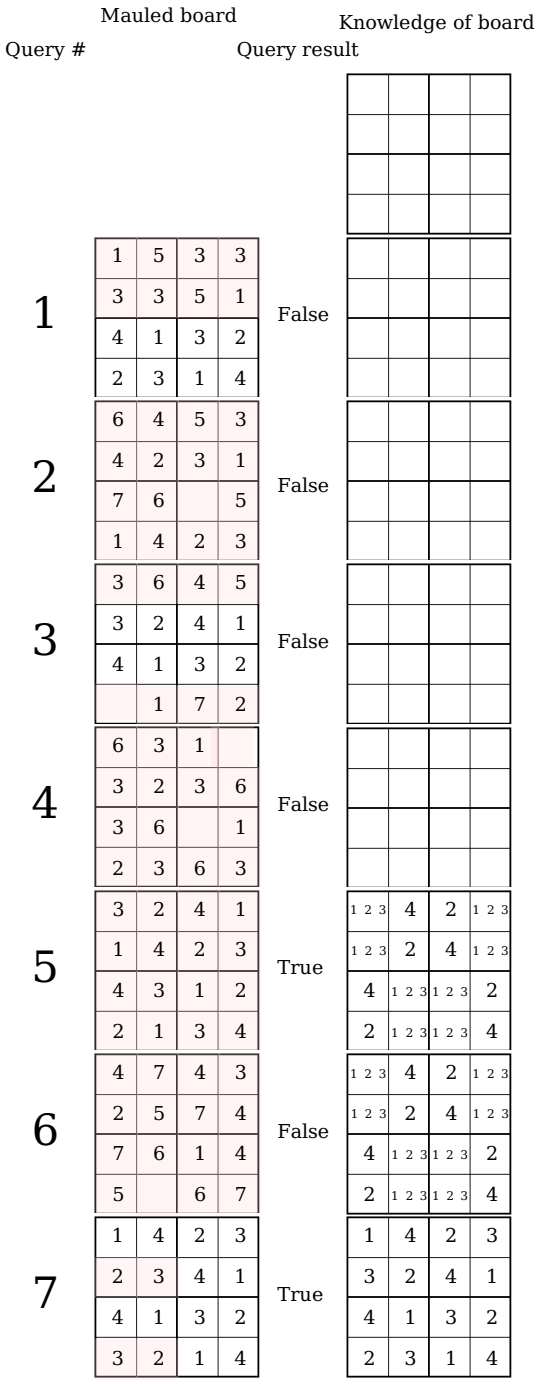


Figure 10: An example of an attack against the Sudoku format check. When the solver has narrowed a square to some candidate entries, multiple numbers are shown in a given square. Squares which are changed from their original values by a malleation are colored in light red.

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. We discuss techniques for evaluating this weighting function in Appendix E.

Plaintext weight information can be encoded via an abstract *plaintext weighting function* $W : \mathcal{M} \to \{0, \ldots, B\}$ defined with respect to some constant B, where $W(M) \to A$ defines the fractional weight $\frac{A}{B}$ of a given plaintext M. A custom weighting function can be developed, for example, to encode the assumption that all PKCS #7 padding lengths are equally likely.

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

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

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

F Format Check Implementations

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

F.1 PKCS #7 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
            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)
```