Automating the Development of Chosen Ciphertext Attacks

Abstract

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

1 Introduction

The past decades have seen enormous improvement in our understanding of cryptographic protocol design. Despite these advances, vulnerable protocols remain widely deployed. In many cases this is a result of continued support for legacy protocols and ciphersuites, such as TLS's CBC-mode ciphers [6, 60], export-grade encryption [3, 8, 17], and legacy email encryption [55]. However, support for legacy protocols does not account for the presence of vulnerabilities in more recent protocols and systems [33, 39, 44, 68, 70].

In this work we consider a specific class of vulnerability: the continued use of unauthenticated symmetric encryption in many cryptographic systems. While the research community has long noted the threat of adaptive-chosen ciphertext attacks on malleable encryption schemes [15, 16, 52], these concerns gained practical salience with the discovery of *padding oracle* attacks on a number of standard encryption protocols [5, 6, 12, 20, 28, 37, 48, 49, 69]. Despite repeated warnings to industry, variants of these attacks continue to plague modern systems, including TLS 1.2's CBC-mode ciphersuite [4, 6, 45] and hardware key management tokens [9, 12]. 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 [39, 42], Apple's iMessage [33] and modern OpenPGP implementations [44, 55]. 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 [19, 40, 43]. 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 [33] report that Apple did not repair a com-

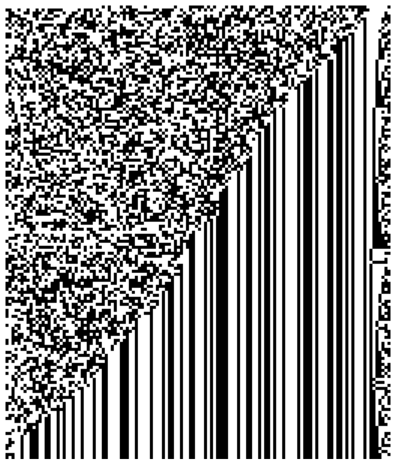


Figure 1: Output of a format oracle attack that our algorithms developed against a bitwise padding check oracle $\mathsf{F}_{\mathsf{bitpad}}$ (see §5.2 for a full description). The original ciphertext is a valid 128-bit (random) padded message encrypted using a stream cipher. Each row of the bitmap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

plex gzip compression format oracle in the iMessage protocol when the lack of authentication was pointed out; but did mitigate the flaw when a concrete exploit was demonstrated. Similar flaws in OpenPGP clients [33, 55] and PDF encryption [51] were addressed only when researchers developed proof-of-concept exploits. The unfortunate aspect of this strategy is that cryptographers' time is limited, which leads protocol designers to discount the exploitability of real cryptographic flaws.

Removing the human element. In this work we investigate the feasibility of *automating the design and development* of adaptive chosen ciphertext attacks on symmetric encryption schemes. We stress that our goal is not simply to automate the execution of known attacks, as in previous works [42]. Instead, we seek to develop a methodology and a set of tools to (1) evaluate if a system is vulnerable to practical exploitation, and (2) programmatically derive a novel exploit strategy, given only a description of the target. This removes the expensive human element from attack development.

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

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

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

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

Relationship to previous automated attack work. Previous work [11, 24, 54] has looked at automatic discovery and exploitation of *side channel* attacks. In this setting, a program combines a fixed secret input with many "low" inputs that are (sometimes adaptively) chosen by an attacker, and produces a signal, *e.g.,* modeling a timing result. This setting can be viewed as a special case of our general model (and vice versa). Like our techniques, several of these works employ SAT solvers and model counting techniques. However, beyond these similarities, there are fundamental differences that manifest in our results: (1) in this work we explore a new approach based on approximate model counting, and (2) as a result of this approach, our results operate over much larger secret domains than the cited works. To illustrate the differences, our experimental results succeed on secret (message) domains of several hundred bits in length, with malleation strings ("low inputs") drawn from similarly-sized domains. By contrast, the cited works operate over smaller secret domains that rarely even reach a size of 2^{24}. Moreover, our format functions are relatively complex. It is an open question to determine whether the experimental results in the cited works can be scaled using our techniques.

Our contributions. In this work we make the following contributions:

- We propose new, and *fully automated* algorithms for developing format oracle attacks on symmetric encryption (and hybrid encryption) schemes. Our algorithms are designed to work with arbitrary format checking functions, using a machine-readable description of the function and the scheme's malleation features to develop the attack strategy.

- We design and implement novel attack-development techniques that use approximate model counting techniques to achieve significantly greater efficiency than previous works. These techniques may be of independent interest.
- We show how to implement our technique practically with existing tools such as SAT and SMT solvers; and propose a number of efficiency optimizations designed to improve performance for specific encryption schemes and attack conditions.
- We develop a working implementation of our techniques using "off-the-shelf" SAT/SMT packages, and provide the resulting software package (which we call `Delphinium`) as an open source tool for use and further development by the research community[1].
- We validate our tool experimentally, deriving several attacks using different format-checking functions. These experiments represent, to our knowledge, the first evidence of a completely functioning end-to-end machine-developed format oracle attack.

1.1 Intuition

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

Describing malleability. Our attacks exploit the malleability characteristics of symmetric encryption schemes. Because the encryption schemes themselves can be complex, we do not want our algorithms to reason over the encryption mechanism itself. Instead, for a given encryption scheme Π, we require the user to develop two efficiently-computable functions that define the malleability properties of the scheme. The function $\mathsf{Maul}^{\Pi}_{\mathsf{ciph}}(C,S) \rightarrow C'$ 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[2]). The essential property we require from these functions is that the plaintext malleation function should "predict" the effects of encrypting a plaintext M, mauling the resulting ciphertext, then subsequently decrypting the result. For some typical encryption schemes, these functions can be simple: for example, a simple stream cipher can be realized by defining both functions to be bitwise exclusive-OR. However, malleation functions may also implement features such as truncation or more sophisticated editing, which could imply a complex and structured malleation string.

Building block: theory solvers. Our techniques make use of efficient theory solvers, such as SAT and Satisfiability Modulo Theories (SMT) [1, 46]. SAT solvers apply a variety of tactics to identify or rule out a satisfying assignment to a boolean constraint formula, while SMT adds a broader range of theories and tactics such as integer arithmetic and string logic. While in principle our techniques can be extended to work with either system, in practice we will focus our techniques to use quantifier-free operations over bitvectors (a theory that easily reduces to SAT). In later sections, we will show how to realize these techniques efficiently using concrete SAT and SMT packages.

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

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

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

[2]This captures the fact that, in some encryption schemes (*e.g.*, CBC-mode encryption), malleation produces *key-dependent* effects on the decrypted message. We discuss and formalize this in §2.

ments chosen by the solver, members of the respective partitions.

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

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

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

Step 0: Initialization. At the beginning of the attack, our attack algorithm receives as input a target ciphertext C^*, as well as a machine-readable description of the functions F and $\mathsf{Maul}_{\mathsf{plain}}$. We require that these descriptions be provided in the form of a constraint formula that a theory solver can reason over. To initialize the attack procedure, the user may also provide an initial constraint predicate $G_0 : \{0,1\}^n \rightarrow \{0,1\}$ that expresses all known constraints over the value of M^*.[3] (If we have no *a priori* knowledge about the distribution of M^*, we can set this initial formula G_0 to be trivial).

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

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

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

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

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

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

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

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

Remark on efficiency. A key feature of the attack described above is that it is *guaranteed* to make progress at each round in which the solver is able to find a satisfying assignment to Equation (1). This is fundamental to the constraint system we construct: our approach forces the solver to ensure that each malleation string S implicitly

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

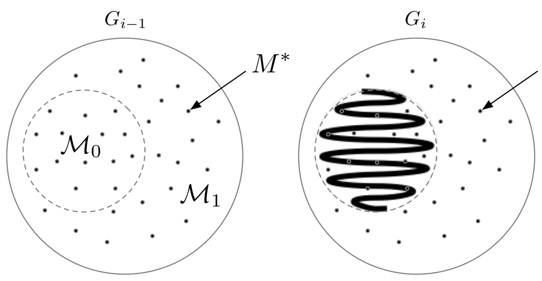


Figure 2: Left: illustration of a plaintext candidate space defined by G_{i-1}, highlighting the two subsets $\mathcal{M}_0, \mathcal{M}_1$ induced by a specific malleation string $\mathbf{S}$. Right: the candidate space defined by G_i, in which many candidates have been eliminated following an oracle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

partitions the candidate message set into a pair (M_0, M_1), such that malleation of messages in either subset by S will produce distinct outputs from the format checking function F. As a consequence of this, for any possible result from the real-world decryption oracle, the updated constraint formula G_i *must* eliminate at least one plaintext candidate that satisfied the previous constraints G_{i-1}.

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

Improving query profitability. We can define the *profitability* $\psi(G_{i-1}, G_i)$ of an experimental query by the number of plaintext candidates that are "ruled out" once an experiment has been executed and the constraint formula updated. In other words, this value is defined as the number of plaintext candidates that satisfy G_{i-1} but do *not* satisfy G_i. The main limitation of our first attack strategy is that it does not seek to optimize each experiment to maximize query profitability.

To address this concern, let us consider a more general description of our attack strategy, which we illustrate in Figure 2. At the i^{th} iteration, we wish to identify a malleation string $\mathbf{S}$ that defines two disjoint subsets $\mathcal{M}_0, \mathcal{M}_1$ of the current candidate plaintext space, such that for any concrete oracle result $\mathbf{r} \in \{0,1\}$ and $\forall M \in \mathcal{M}_{\mathbf{r}}$ it holds that $\mathsf{F}(\mathsf{Maul}_{\mathsf{plain}}(M, \mathbf{S})) = \mathbf{r}$. In this description, any concrete decryption oracle result must "rule out" (at a minimum) every plaintext contained in the subset $\mathcal{M}_{1-\mathbf{r}}$. This sets $\psi(G_{i-1}, G_i)$ equal to the cardinality of $\mathcal{M}_{1-\mathbf{r}}$.

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

Technical challenge: model count optimization. While our new formulation is conceptually simple, actually realizing it involves overcoming serious limitations in current theory solvers. This is due to the fact that, while several production solvers provide optimization capabilities [46], 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).[4]

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 [65, 66], and practical counting algorithms solutions [13, 18] tend to perform poorly when the combinatorial space is large and the satisfying assignments are sparsely distributed throughout the space, a condition that is likely in our setting. The specific optimization problem our techniques require proves to be even harder. Indeed, only recently was such a problem formalized, under the name $\mathsf{Max\#SAT}$ [32].

Approximating $\mathsf{Max\#SAT}$. While an exact solution to $\mathsf{Max\#SAT}$ is $\mathsf{NP}^{\mathsf{PP}}$-complete [32, 65], several works have explored *approximate* solutions to this and related counting problems [23, 32, 34, 61]. One powerful class of approximate counting techniques, inspired by the theoretical work of Valiant and Vazirani [67] and Stockmeyer [63], 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

[4]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.

of satisfying assignments approximately by half, independently of the underlying distribution of valid solutions. The implication is as follows: if a satisfying assigment to the enhanced formula exists, we should be convinced (probabilistically) that the original formula is likely to possess on the order of 2^s satisfying assignments. Subsequently, researchers in the model counting community showed that with some refinement, these approximate counting strategies can be used to approximate Max#SAT [32], although with an efficiency that is substantially below what we require for an efficient attack.

To apply this technique efficiently to our attack, we develop a custom count-optimization procedure, and apply it to the attack strategy given in the previous section. At the start of each iteration, we begin by conjecturing a candidate set size 2^s for some non-negative integer s, and then we query the solver for a solution to (S, M_0, M_1) in which approximately 2^s solutions can be found for *each* of the abstract bitvectors M_0, M_1. This involves modifying the equation of Step (1) by adding s random parity constraints to *each* of the abstract representations of M_0 and M_1. We now repeatedly query the solver on variants of this query, with increasing (resp. decreasing) values of s, until we have identified the maximum value of s that results in a satisfying assignment[5]. 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.* [34], we are able to substantially improve our certainty by conducting t *trials* within each query, accepting iff at least $\lceil(\frac{1}{2}+\delta)t\rceil$ trials are satisfied, where δ is an adjustable tolerance parameter.

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, 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

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 use $\mathcal{M}$ to denote the set of valid plaintexts accepted by a scheme, and $\mathcal{C}$ to denote the set of valid ciphertexts.

2.1.1 Malleation Functions

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

Definition 2 (Malleation functions) The *malleation functions* for a symmetric encryption scheme Π comprise a pair of efficiently-computable functions $(\mathsf{Maul}^{\Pi}_{\mathsf{ciph}}, \mathsf{Maul}^{\Pi}_{\mathsf{plain}})$ with the following properties. Let $\mathcal{M}, \mathcal{C}$ be the plaintext (resp. ciphertext) space of Π. The function $\mathsf{Maul}^{\Pi}_{\mathsf{ciph}} : \mathcal{C} \times \{0,1\}^* \rightarrow \mathcal{C} \cup \{\perp\}$ takes as

[5] 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.

input a ciphertext and a *malleation instruction string*. It outputs a ciphertext or the distinguished error symbol $\perp$. The function $\mathsf{Maul}^{\Pi}_{\mathsf{plain}} : \mathcal{M} \times \{0,1\}^* \rightarrow \hat{\mathcal{M}}$, on input a plaintext and a malleation instruction string, outputs a *set* $\hat{\mathcal{M}} \subseteq \mathcal{M} \cup \{\perp\}$ of possible plaintexts (augmented with the decryption error symbol $\perp$). The structure of the malleation string is entirely defined by these functions; since our attack algorithms will reason over the functions themselves, we treat S itself as an opaque value.

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 $\mathsf{Maul}^{\Pi}_{\mathsf{ciph}}(C,S) \neq \perp$:

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

In Appendix C, we discuss a collection of encryption schemes and their associated malleation functions.

2.2 Theory Solvers and Model Counting

Solvers take as input a system of constraints over a set of variables, and attempt to derive (or rule out the existence of) a satisfying solution. Modern SAT solvers generally rely on two main families of theorem solver: DPLL [26, 27] and Stochastic Local Search [36]. 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[6]. For the purposes of describing our algorithms, we specify a query to the solver by the subroutine $\mathsf{SATSolve}\{(A_1,\ldots,A_N) : G\}$ where $A_1,\ldots,A_N$ each represent abstract bitvectors of some defined length, and G is a constraint formula over these variables. The response from this call provides one of three possible results: (1) sat, as well as a concrete satisfying solution $(\mathbf{A_1}\ldots,\mathbf{A_N})$, (2) the distinguished response unsat, or (3) the error $\mathsf{unknown}$.

Model counting and Max#SAT. While SAT determines the existence of a single satisfying assignment, a more general variant of the problem, #SAT, determines the *number* of satisfying assignments. In the literature this problem is known as *model counting* [10, 13, 18, 22, 34, 59, 66, 71].

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

In this work we make use of a specific optimization variant of the model count problem, which was formulated as Max#SAT by Fremont *et al.* [32]. In a streamlined form, the problem can defined as follows: given a boolean formula $\phi(X,Y)$ over abstract bitvectors X and Y, find a concrete assignment to X that *maximizes* the number of possible satisfying assignments to Y.[7] We will make use of this abstraction in our attacks, with realizations discussed in §3.2. Specifically, we define our main attack algorithm in terms of a generic Max#SAT oracle that has the following interface:

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

2.3 Format Checking Functions

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

Function descriptions. Requiring format checking functions to be usable within SAT/SMT solvers raises additional implementation considerations. Refer to the full version of the paper for discussion of these considerations, and to Appendix E for examples.

3 Our Constructions

In this section we present a high-level description of our main contribution: a set of algorithms for programmatically conducting a format oracle attack. First, we provide pseudocode for our main attack algorithm, which uses a generic Max#SAT oracle as its key ingredient. This first algorithm can be realized approximately using techniques such as the MaxCount algorithm of Fremont *et al.* [32], although this realization will come at a significant cost to practical performance. To reduce this cost and make our attacks practical, we next describe a concrete replacement algorithm that can be used in place of a Max#SAT solver. The combination of these algorithms forms the basis for our tool `Delphinium`.

[7]The formulation of Fremont *et al.* [32] includes an additional set of boolean variables Z that must also be satisfied, but is not part of the optimization problem. We omit this term because it is not used by our algorithms. Note as well that, unlike Fremont *et al.*, our algorithms are not concerned with the actual *count* of solutions for Y.

3.1 Main Algorithm

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

This algorithm largely follows the intuition described in §1.1. At each iteration, it derives a concrete malleation string $\mathbf{S}$ using the Max#SAT oracle in order to find an assignment that maximizes the number of solutions to the abstract bitvector $M_0 \| M_1$. It then mauls C^* using this malleation string, and queries the decryption oracle $\mathcal{O}_{\mathsf{dec}}$ on the result. It terminates by outputting a (possibly incomplete) description of M^*. This final output is determined by a helper subroutine `SolveForPlaintext` that uses the solver to find a unique solution for M^* given a constraint formula, or else to produce a human-readable description of the resulting model.[8]

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

A proof of Theorem 3.1 appears in Appendix A.

Remarks. Note that a greedy adaptive attack may not be globally optimal. It is hypothetically possible to modify the algorithm, allowing it to reason over multiple oracle queries simultaneously (in fact, Phan *et al.* discuss such a generalization in their side channel work [54]). We find that this is computationally infeasible in practice. Finally, note also that our proof assumes an exact Max#SAT oracle. In practice, this will likely be realized with a probably approximately correct instantiation, causing the resulting attack to be a probably approximately greedy attack.

3.2 Realizing the Max#SAT Oracle

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

Realization from Fremont *et al.* Fremont *et al.* [32] propose an approximate algorithm called MaxCount that can be used to instantiate our attack algorithms. The MaxCount algorithm is based on repeated application of approximate counting and sampling algorithms [21–23], which can in turn be realized using a general SAT solver. While MaxCount is approximate, it can be tuned to provide a high degree of accuracy that is likely to be effective for our attacks. Unfortunately, the Fremont *et al.* solution has two significant downsides. First, to achieve the discussed bounds requires parameter selections which induce infeasible queries to the underlying SAT solver. Fremont *et al.* address this by implementing their algorithm with substantially reduced parameters, for which they demonstrate good empirical performance. However, even the reduced Fremont *et al.* approach still requires numerous calls to a solver. Even conducting a single approximate count of solutions to the constraint systems in our experiments could take hours to days, and such counts might occur several times in a single execution of MaxCount.

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

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

The goal of this approach is to identify a malleation string $\mathbf{S}$ as well as the largest integer s such that at least 2^s solutions can be found for each of M_0, M_1. To improve the accuracy of this approach, we employ a technique originally pioneered by Gomes *et al.* [34] and modify each SAT query to include multiple *trials* of this form, such that only a fraction $\delta + 1/2$ of the trials must succeed in order for $\mathbf{S}$ to be considered valid. The parameters t, δ are adjustable; we evaluate candidate values in §5.

Unlike Fremont *et al.* (at least, when implemented at full parameters) our algorithm does not constitute a

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

sound realization of a Max#SAT solver. However, empirically we find that our attacks using FastSample produce query counts that are close to the optimal possible attack. More critically, our approach is capable of identifying a candidate malleation string in seconds on the constraint systems we encountered during our experiments.

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

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

The input bit b_{success} indicates whether or not a solution was found for a conjectured size s, while n provides a known upper-bound. The history string $Z \in \{0,1\}^*$ allows the routine to record state between consecutive calls. `AdjustSize` outputs a bit b_{continue} indicating whether the attack should attempt to find a new solution, as well as an updated set size s'. If `AdjustSize` is called with $s = 0$, then s' is set to an initial set size to test, $b_{\mathsf{continue}} = \text{TRUE}$, and $Z' = Z$.

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

Algorithm 1: DeriveAttack

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

Algorithm 2: FastSample

```
Input: $\phi$ a constraint system over abstract
       bitvectors $S, M_0 \| M_1$; $n$ the maximum
       length of (each of) $M_0, M_1$; $m$ the
       maximum length of $S$; $t$ number of trials; $\delta$
       fraction of trials that must succeed
Output: $\mathbf{S} \in \{0,1\}^m$
Procedure:
$b_{\mathsf{continue}} \leftarrow$ TRUE;
$Z$, $\mathbf{S}$, $s' \leftarrow \perp, \varepsilon, 0$;
$s \leftarrow$ AdjustSize(FALSE, $n, \perp, Z$);
// define $t$ symbolic copies of the
   abstract bitvectors $M_0$, $M_1$, and a
   new constraint system $\phi^t$
$\{M_{1,0}, \dots, M_{t,0}\} \leftarrow M_0$;
$\{M_{1,1}, \dots, M_{t,1}\} \leftarrow M_1$;
Define $\phi^t(S, \{M_{1,0}, \dots, M_{t,0}\}, \{M_{1,1}, \dots, M_{t,1}\})$ as
$\phi(S, M_{1,0} \| M_{1,1}) \wedge \dots \wedge \phi(S, M_{t,0} \| M_{t,1})$;
while $b_{\mathsf{continue}}$ do
    // Construct $2st$ parity constraints
    for $i \leftarrow 1$ to $t$ do
        $\mathcal{H}_{i,0} \leftarrow$ ParityConstraint $(n, s)$;
        $\mathcal{H}_{i,1} \leftarrow$ ParityConstraint $(n, s)$
    // Query the solver
    $\mathbf{S} \leftarrow$
    SATSolve$\{(S, \{M_{1,0}, \dots, M_{t,0}\}, \{M_{1,1}, \dots, M_{t,1}\})$ :
        $\exists \mathcal{R}_0 \subseteq [1,t] : |\mathcal{R}_0| \geq$
        $\lceil (0.5 + \delta) t \rceil, \forall j \in \mathcal{R}_0 : \mathcal{H}_{j,0}(M_{j,0}) = 1 \wedge$
        $\exists \mathcal{R}_1 \subseteq [1,t] : |\mathcal{R}_1| \geq$
        $\lceil (0.5 + \delta) t \rceil, \forall j \in \mathcal{R}_1 : \mathcal{H}_{j,1}(M_{j,1}) = 1 \wedge$
            $\phi^t(S, \{M_{1,0}, \dots, M_{t,0}\}, \{M_{1,1}, \dots, M_{t,1}\})\}$;

    if $S ==$ unsat then
        $b_{\mathsf{success}} =$ FALSE;
    $(b_{\mathsf{continue}}, s, Z) \leftarrow$
    AdjustSize($b_{\mathsf{success}}, n, s, Z$);
return $\mathbf{S}$
```

4 Prototype Implementation

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

4.1 Architecture Overview

Figure 3 illustrates the architecture of `Delphinium`. The software comprises several components:

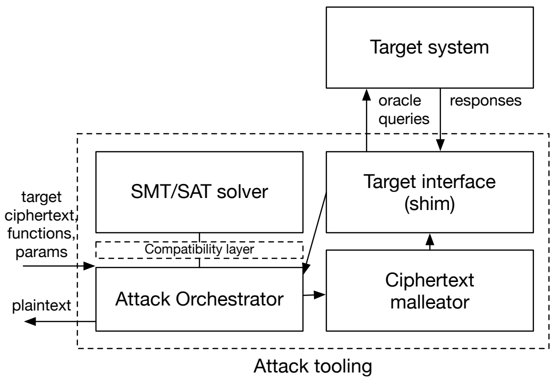


Figure 3: Architecture of `Delphinium`.

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

SMT/SAT solver. Our implementation supports multiple SMT solver frameworks (STP [1] and Z3 [46]) 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.
As part of our prototype implementation, we provide working examples for each of these modules, as well as a test harness to evaluate attacks locally.

4.2 Implementation Details

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

Selecting SAT and SMT solvers. In the course of this work we evaluated several SMT and SAT solvers optimized for different settings. Seeking the best of a few worlds, we use Z3 for formula manipulation and CryptoMiniSAT as a solving backend, bridged by CNF formula representations. Refer to the full version of this paper for discussion and challenges of the solvers we evaluated.

Low-density parity constraints. Our implementation of model counting requires our tool to incorporate $2st$ distinct parity functions into each solver query. Each parity constraint comprises an average of $\frac{n}{2}$ exclusive-ORs (where n is the maximum length of M^*), resulting in a complexity increase of tens to hundreds of gates in our SAT queries. To address this, we adopted an approach used by several previous model counting works [29, 73]: 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 compuationally expensive, we evaluate a few strategies for implementing `AdjustSize` which minimize time spent solving. We omit discussion of these strategies for brevity; refer to the full version of this paper.

Describing malleation. To avoid making users re-implement basic functionality, `Delphinium` provides built-in support for several malleation functions. These include simple stream ciphers, stream ciphers that support truncation (from either the left or the right side), and CBC mode encryption. The design of these malleation functions required substantial extensions to the `Delphinium` framework. For reasons of space we provide this discussion in Appendix C.

Test Harness. For our experiments in Section 5 we developed a test harness to implement the Ciphertext Malleator and Target Interface shim. This test harness implements the code for mauling and decrypting M^* locally using a given malleation string $\mathbf{S}$.

4.3 Software

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

4.4 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 [31, 50]. 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. We additionally provide a translation tool from the output format of CMBC-GC [31] to Python (entirely comprised of circuit operations) to enable use of the Python front-end to `Delphinium`.

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[9]. Several additional tests were run a 72-core Intel Xeon E5 CPU with 500GB of memory running on Ubuntu 16.04, and a 96-core Intel Xeon E7 CPU with 1TB of memory running Ubuntu 18.04. We refer to these machines as AWS, E5 and E7 in the sections below.

Data collection. For each experimental run, we collected statistics including the total number of decryption oracle queries performed; the wall-clock time required to construct each query; the number of plaintext bits recovered following each query; and the value of s used to construct a given malleation string. We also recorded each malleation string $\mathbf{S}$ produced by our attack, which allows us to "replay" any transcript after the fact. The total number of queries required to complete an attack provides the clearest signal of attack progress, and we use that as the primary metric for evaluation. However, in some cases we evaluate partial attacks using the `ApproxMC` approximate model counting tool [61]. This tool provides us with an estimate for the total number of remaining candidates for M^* at every phase of a given attack, and thus allows evaluation of partial attack transcripts.

Parameter Tuning. The adjustable parameters in `FastSample` include t, the number of counting trials, δ, which determines the fraction of trials that must succeed, and the length of the parity constraints used to sample. We ran a number of experiments to determine optimal values for these parameters across the format functions PKCS7 and a bitwise format function defined in 5.2. Empirically, $\delta = 0.5$, $2 \leqslant t \leqslant 5$, and hash functions of logarthmic length are suitable for our purposes. More details are in Appendix B.2.

5.2 Experiments with Stream Ciphers

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

Bytewise Encryption Padding. The PKCS #7 encryption standard (RFC 2315) [41] defines a padding scheme for use with block cipher modes of operation. This padding is similar to the standard TLS CBC-mode padding [6] considered by Vaudenay [69]. We evaluate our algorithm on both these functions as a benchmark because PKCS7 and its variants are reasonably complex, and because the human-developed attack is well understood. Throughout the rest of this paper, we refer to PKCS7 and TLS-PKCS7 for sake of clarity.

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 $G_0 \leftarrow \mathsf{F}_{\mathsf{PKCS7}}$.[10]

Results. Running on E5, our four complete attacks completed in an average of 1699.25 queries (min. 1475, max. 1994) requiring 1.875 hours each (min. 1.63, max. 2.18). A visualization of the resulting attack appears in the Appendices. These results compare favorably to the Vaudenay attack, which requires ~2000 queries in expectation, however it is likely that additional tests would find some examples in excess of this average. As points of comparison, attacks with $t = 3$ resulted in a similar number of queries (modulo expected variability over different randomly sampled messages) but took roughly 2 to 3 times

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

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

as long to complete, and attacks with $t = 1$ reached over 5000 queries having only discovered half of the target plaintext message.

Bitwise Padding. To test our attacks, we constructed a simplified *bit* padding scheme $\mathsf{F}_{\mathsf{bitpad}}$. This contrived scheme encodes the bit length of the padding P into the rightmost $\lceil log_2(n) \rceil$ bits of the plaintext string, and then places up to P padding bits directly to the left of this length field, with each padding bit set to 1. We verified the effectiveness of our attacks against this format using a simple stream cipher. Using the parameters $t = 5$, $\delta = 0.5$ the generated attacks took on average 153 queries (min. 137, max. 178). Figure 1 shows one attack transcript at $t = 5, \delta = 0.5$, and Figure 6 shows attack runs with varying values for t.

Negative result: Cyclic Redundancy Checks (CRCs). Cyclic redundancy checks (CRCs) are used in many network protocols for error detection and correction. CRCs are well known to be malleable, due to the linearity of the functions: namely, for a crc it is always the case that $\text{CRC}(a \oplus b) = \text{CRC}(a) \oplus \text{CRC}(b)$. To test `Delphinium`'s ability to *rule out* attacks against format functions, we implemented a message format consisting of up to three bytes of message, followed by a CRC-8 and a 5-bit message length field. The format function $\mathsf{F}_{\mathsf{crc8}}$ computes the CRC over the message bytes, and verifies that the CRC in the message matches the computed CRC.[11] A key feature of this format is that a valid ciphertext C^* should *not* be vulnerable to a format oracle attack using a simple exclusive-OR malleation against this format, for the simple reason that the attacker can predict the output of the decryption oracle for every possible malleation of the ciphertext (due to the linearity of CRC), and thus no information will be learned from executing a query. This intuition was confirmed by our attack algorithm, which immediately reported that no malleation strings could be found.

5.3 Ciphers with Truncation

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

CRC-8 with a truncatable stream cipher. To evaluate how truncation affects the ability of `Delphinium` to find attacks, we conducted a second attack using the function $\mathsf{F}_{\mathsf{crc8}}$, this time using an implementation of AES-CTR supporting truncation. Such a scheme may seem contrived, since it involves an encrypted CRC value. However, this very flaw was utilized by Beck and Trew to break WPA [64]. In our experiment, the attack algorithm was able to recover two bytes of the three-byte message, by using the practical strategy of truncating the message and iterating through all possible values of the remaining byte. A list of the malleation strings produced by the attack appears in Figure 9.

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

S2N with Exclusive-OR, Truncation. To evaluate a realistic attack on a practical format function, we developed a format checking function for the Amazon `s2n` [2] 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 [19, 30].

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

5.4 CBC mode

We also used the malleation function for CBC-mode encryption. This malleation function supports an arbitrary number of blocks, and admits truncation of plaintexts

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

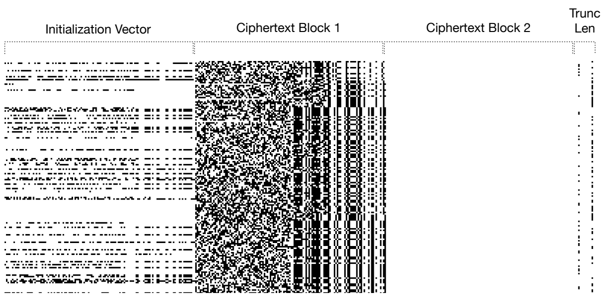


Figure 4: A contiguous set of malleation queries made by `Delphinium` during a simulated CBC attack. The rightmost bits signal truncation (from left or right).

by an arbitrary number of *blocks* from either side of the plaintext.[12] The CBC malleation function accepts a structured malleation string S, which can be parsed as (S', l, r) where l, r are integers indicating the number of blocks to truncate from the message.

To test this capability, we used the PKCS7 format function with a blocksize of $B = 16$ bytes, and a two-block CBC plaintext. (This corresponds to a ciphertext consisting of three blocks, including the Initialization Vector.) `Delphinium` generated an attack which took 3441 oracle queries for a random message with four bytes of padding. This compares favorably to the Vaudenay attack, which requires 3588 queries in expectation. Interestingly, `Delphinium` settled on a more or less random strategy of truncation. Where a human attacker would focus on recovering the entire contents of one block before truncating and attacking the next block of plaintext, `Delphinium` instead truncates more or less as it pleases: in some queries it truncates the message and attacks the Initialization Vector. In other queries it focuses on the second block. Figure 4 gives a brief snapshot of this pattern of malleations discovered by `Delphinium`. Despite this query efficiency (which we seek to optimize, over wall-clock efficiency), the compute time for this attack was almost a week of computation.

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 [15, 57, 58], or analyzing deployed protocols, *e.g.,* [14]. Among many practical format oracle attacks [9, 12, 33, 39, 42, 44, 53, 55, 56, 72], the Lucky13 attacks [4, 6] 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 [24], [54] focuses on generating public input values that lead to maximum leakage of secret input in Java programs where leakage is defined in terms of channel capacity and shannon entropy. Unlike our work, Pasareanu et al. [24] do not consider an adversary that makes *adaptive* queries based on results of previous oracle replies. Both [24] and [54] assume leakage results from timing and memory usage side channels.

Using solvers for cryptographic tasks/model counting. A wide variety of cryptographic use cases for theory solvers have been considered in the literature. Soos *et al.* [62] developed CryptoMiniSAT to recover state from weak stream ciphers, an application also considered in [25]. Solvers have also been used against hash functions [47], and to obtain cipher key schedules following cold boot attacks [7]. There have been many model counting techniques proposed in the past based on universal hash functions [34, 73]. However, many other techniques have been proposed in the literature. Several works propose sophisticated multi-query approach with high accuracy [23, 61], resulting in the `ApproxMC` tool we use in our experiments. Other works examine the complexity of parity constraints [73], and optimize the number of variables that must be constrained over to find a satisfying assignment [38].

7 Conclusion

Our work leaves a number of open problems. In particular, we proposed several optimizations that we were not able to implement in our tool, due to time and performance constraints. Additionally, while we demonstrated the viability of our model count optimization techniques through empirical analysis, these techniques require theoretical attention. Our ideas may also be extensible in many ways: for example, developing automated attacks on protocols with side-channel leakage; on public-key encryption; and on "leaky" searchable encryption schemes, *e.g.,* [35]. 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.

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

References

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

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

[3] 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.

[4] 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.

[5] Martin R. Albrecht, Kenneth G. Paterson, and Gaven J. Watson. Plaintext recovery attacks against SSH. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, SP '09, pages 16–26, Washington, DC, USA, 2009. IEEE Computer Society.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[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.

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

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

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

[22] 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.

[23] 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.

[24] 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.

[25] 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.

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

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

[28] Jean Paul Degabriele and Kenneth G. Paterson. On the (in)security of IPsec in MAC-then-encrypt configurations. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 493–504, New York, NY, USA, 2010. ACM.

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

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

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

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

[37] 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.

[38] 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.

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

[40] 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.

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

[42] 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.

[43] 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.

[44] 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.

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

[46] Microsoft Research. The Z3 Theorem Prover. Available at https://github.com/Z3Prover/z3.

[47] 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.

[48] Chris J. Mitchell. Error oracle attacks on CBC mode: Is there a future for CBC mode esncryption? In *Information Security, 8th International Conference, ISC 2005, Singapore, September 20-23, 2005, Proceedings*, pages 244–258, 2005.

[49] 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.

[50] 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.

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

[52] 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.

[53] 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.

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

[55] 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.

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

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

[58] 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.

[59] 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.

[60] 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.

[61] 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.

[62] Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT Solvers to Cryptographic Problems. In *International Conference on Theory and Applications of Satisfiability Testing (SAT 2009)*, pages 244–257, 06 2009.

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

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

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

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

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

[68] 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.

[69] 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.

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

[71] 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.

[72] 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.

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

A Proof that DeriveAttack is Greedy

A.1 Preliminaries

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

Let M be the distribution of plaintext messages of some fixed length n. We consider the uniform message distribution, but non-uniform distributions are equivalently supported through weighted model counting (which reduces to unweighted model counting efficiently) [22]. The message length is known to the attack algorithm.

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

Let G_0 be the initial constraint system, a conjuction of the following constraints:

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

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

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

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

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

A.2 Proof

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

A.2.1 Base Case

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

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

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

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

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

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

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

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

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

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

A.2.2 Strong Induction

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

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

A.2.3 Max#SAT Aborts

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

B Additional Experimental Results

B.1 PKCS #7

Figure 5 demonstrates an example sequence of malleations strings displayed as bitmaps, with one malleation string per row. This image represents a full attack, constituted of 1475 queries, against PKCS #7 with a 128-bit plaintext encrypted with a stream cipher. As the attack progresses, bytes are discovered from right to left in a similar fashion to the human attack. This emergent behavior occurs as it is the most efficient way to discover the underlying plaintext bytes, in expectation.

B.2 Bitwise Padding Format Tests Over *t*

We empirically observe the parameter δ has limited impact on solver runtime and set $\delta = 0.5$[13]. This produces reasonable performance and is consistent with previous works [34]. After running a variety of test we determined that t is a key parameter affecting attack runtime. Analysis confirmed that larger t increases query profitability by increasing the certainty of the underlying counting formula [34]. However, larger t significantly increases the complexity of each constraint formula, which results in an (approximately linear) increase in CNF complexity and a variable increase in solver runtime depending on the underlying constraints. Additionally, despite increased runtime, we find that increasing t is subject to rapidly diminishing returns. As such, we generally use the lowest value for t which empirically succeeds.

For example, to evaluate the impact of changing t on our attacks against the bytewise PKCS #7 padding format, we conducted a series of "microbenchmarks" of our attack's experiment generation procedure. In each experiment, we asked the algorithm to derive 10 malleation strings **S** at arbitrary points along a complete attack run, and then used the `ApproxMC` model counting tool to measure and average the minimum profitability of the resulting malleation string[14]. We repeated each experiment for each $t \in \{1, \ldots, 6\}$, and measured the wall-clock execution time of experiment generation. This strategy is necessarily heuristic, but is a useful method for indicat-

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

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

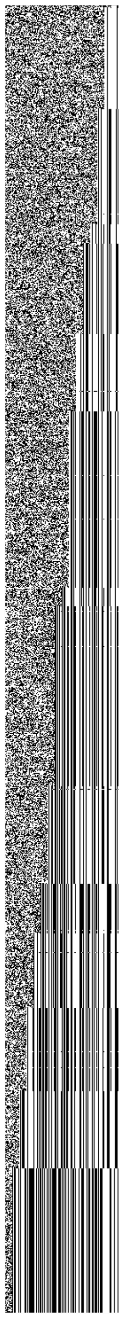


Figure 5: PKCS #7 malleation bitmap

ing appropriate parameters for a full attack run without *a priori* knowledge of attack performance.

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 a bitwise padding format described in further detail in 5.2.

Figure 6 demonstrates malleation string bitmaps for full attacks against the bitwise padding format for $t \in 2,\ldots,5$. The $t = 1$ bitmap is truncated significantly.

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 PKCS #7 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.

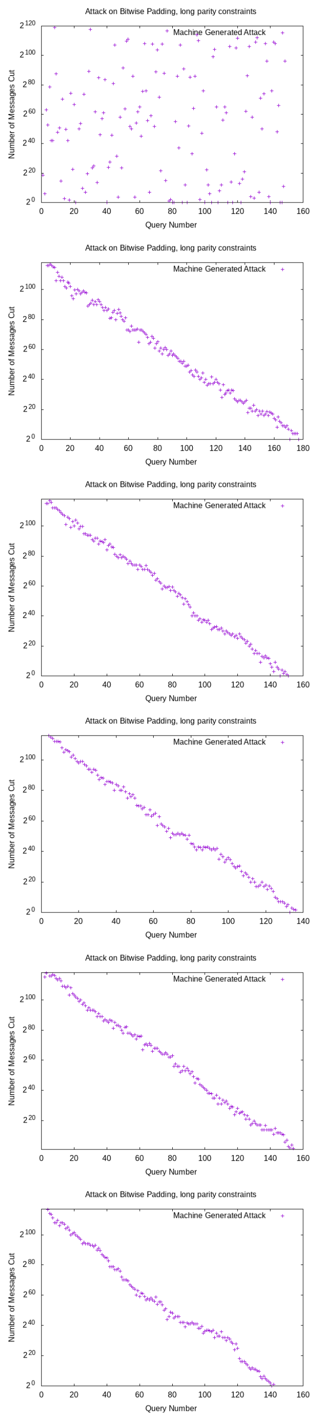


Figure 7: Approximations of how many messages were removed in a 128-bit Bitwise Padding Format attack with long parity constraints.

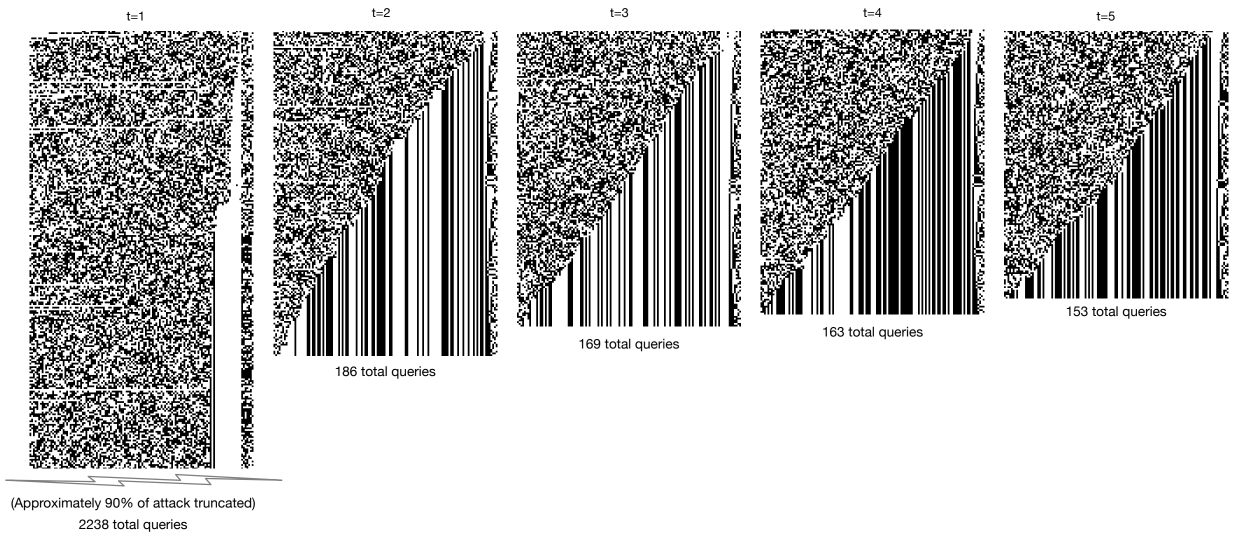
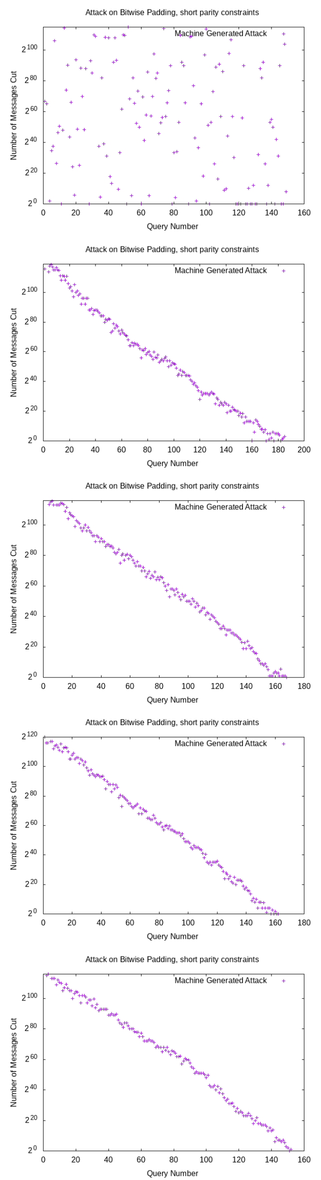


Figure 6: Malleation strings for 128-bit bitwise padding ($\mathsf{F}_{\mathsf{bitpad}}$) using a stream cipher. Experiments consider several different values of t, all with $\delta = 0.5$. Each uses a different random 128-bit message, which can account for some variability in the attack progress. For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



B.3 CRC

Figure 9 displays a bitmap of malleation strings for an attack against CRC, allowing truncation and exlucsive-OR malformations.

C Implementing Malleation Functions

Truncation. Support for truncation requires `Delphinium` to support plaintexts of variable length. This functionality is not natively provided by the bitvector interfaces used in most solvers. We therefore modify our variables to encode both message content, as well as length. This necessitates changes to the interface for F. We accomplish this by treating the first $log_2(n)$ bits of each bitvector as a *length field* specifying how long the message is and by having every implementation of F decode this value prior to evaluating the plaintext. To properly capture truncation off either end of a message, the malleation bitvector is extended by $2\dot{l}og_2(n)$ where the lowest order $log_2(n)$ bits specify how many bits should be truncated off the low order bits of the plaintext and the other $log_2(n)$ bits specify what should be truncated from high order bits of the message. For ease of implementation, the first n bits of the malleation string are also extended to cover the length field of the plaintext. This allows for easily expressing the exclsuive-OR portion of our malleation

Figure 9: Representation of a malleation string an attack on a CRC-8 where the length of the data to compute the CRC over is 3 bytes long.

without bitshifting and allows encoding extension. As stream ciphers only support arbitrary truncation off one side of the message, we add a constraint to the boolean SAT formula which disallows truncation off the low order bits of a message.

Truncation for CTR/OFB Block Cipher Modes. Since CTR and CFB operate as pseudo-stream ciphers, the exclusive-OR implementation for regular stream ciphers suffices to describe exclusive-or malleability for these block cipher modes. We also have the ability to increment the nonce and thus truncate off early blocks of ciphertext, which is not possible when dealing with ciphertext encrypted using other stream ciphers. To capture these capabilities, we relax the restriction on the malleation to allow for truncation off the left of a message, provided the number of bits being truncated is a multiple of the block size.

CBC Block Cipher Mode. In contrast to a stream cipher, $\mathsf{Maul}^{\Pi_{\mathsf{CBC}}}_{\mathsf{plain}}$ is not equal to $\mathsf{Maul}^{\Pi_{\mathsf{CBC}}}_{\mathsf{ciph}}$ and moreover $\mathsf{Maul}^{\Pi_{\mathsf{CBC}}}_{\mathsf{plain}}$ is significantly more complex. In CBC mode, decryption of a ciphertext block C_i is defined as $P_i = Dec_k(C_i) \oplus C_{i-1}$ where C_{i-1} denotes the previous ciphertext block. Since the block C_i is given directly to a block cipher, any implementation must account for the the fact that any modification of the block C_i creates an unpredictable effect on the output P_i. If we want a SAT/SMT solver to reason over this, we need to include in our constraint formula clauses corresponding to encryption and decryption of messages. To avoid this overhead, we instead 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. When $\mathsf{Mask}[j] = 0$, the value of the output message is equal to $M[j]$ at that position, and when $\mathsf{Mask}[i] = 1$ the value at position $M[j]$ must be viewed as unconstrained.

This requires that we modify F to take (M, Mask) as input. The format function must then reason over both M and Mask. Finally, the output of F is modified 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.[15] Realizing this formulation requires only minor implementation changes to our core algorithms.

Exclusive-OR and Truncation for CBC. With CBC mode decryption, manipulating a preceding ciphertext block C_{i-1} produces a predictable exclusive-or into the plaintext block P_i. A message that has been encrypted with a block cipher can also be reasonably truncated, provided that truncation is done on the order of ciphertext blocks. Therefore, we define malleability that captures (1) blockwise truncation (from either the left or right side of the ciphertext), and (2) exclusive-OR where exclusive-OR at index i in one block produces the same exclusive-OR at index i in the next block of decrypted ciphertext. We keep the encoding of the malleation string the same as in the earlier schemes which support truncation and exclusive-OR, the only difference is that the beginning malleation field does not include a length parameter. We instead enforce this value to be less than the truncated message so that a message cannot be extended by adding another clause to the initial constraints for our attack algorithm.

Supporting Extension by Unknown Plaintext. For encryption schemes that allow truncation off of the beginning of a message it may be helpful to allow the attacker the ability to extend a ciphertext, even if the ciphertext is being extended by unknown plaintext. Specifically this is the case when a format function requires messages of a particular length but bits of the message are never touched by the format function. Because of the breakdown between mauling plaintext and mauling ciphertext, implementing extension naively can result in an attacker

[15]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.

extending a message by plaintext it knows to be correctly formatted, which is not possible in practice. We therefore implemented a subset of this functionality by only allowing truncation such that extension will cover bits that are not checked by the format function.

D Realizing Plaintext Weight Functions

As discussed in §3, our basic attack algorithm is designed to identify the *largest subset* of messages to be eliminated at the i^{th} query. From an optimization perspective, this strategy embeds a critical assumption: namely, that every valid plaintext admitted by G_i is equally likely, and thus the most efficient path to finding M^* is to simply eliminate as many messages as possible, without regard to message "quality".

This approach can produce counterintuitive results in practice. One example manifests in our attacks on the PKCS #7 encryption padding scheme [41], which we discuss in §5. In this format, messages may be padded with up to P bytes of padding. If our target is a valid ciphertext C^* in which $\mathsf{F}(M^*) = 1$, then plaintext candidates with $P = 1$ bytes of padding will be 2^8 times as common as candidates with $P = 2$ bytes of padding, and so on. Since our attack strategy is optimized to eliminate the *largest subset* of messages first, experiments that eliminate longer messages will tend to dominate over those that eliminate short messages. A practical result is that our attack algorithm will "assume" that short padding is more likely than longer padding, and will focus on experiments that make sense based on this assumption. Such a strategy makes sense if a typical M^* is truly 2^{112} times more likely to have $P = 1$ than to have $P = 15$. However, such a distribution may not reflect the typical distribution of messages in a real system.

Approximations with plaintext weight. To address this assumption, our attack algorithm can be updated to *weight* plaintext candidates according to some formula provided by the user. We discuss techniques for evaluating this weighting function in Appendix D.

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 $W(M) = B$.

E Format Check Implementations

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

E.1 PKCS #7 padding $\mathsf{F}_{\mathsf{PKCS7}}$

```python
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
```

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

```python
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
```

E.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)
```