Synthesis of Adaptive Side-Channel Attacks

Quoc-Sang Phan*, Lucas Bang†, Corina S. Păsăreanu*‡, Pasquale Malacaria§, Tevfik Bultan†
*Carnegie Mellon University, Moffett Field, CA 94035, USA
†University of California at Santa Barbara, Santa Barbara, CA 93106, USA
‡NASA Ames Research Center, Moffett Field, CA 94035, USA
§Queen Mary University of London, London E1 4NS, UK

***Abstract*—We present symbolic analysis techniques for detecting vulnerabilities that are due to adaptive side-channel attacks, and synthesizing inputs that exploit the identified vulnerabilities. We start with a *symbolic attack model* that encodes succinctly all the side-channel attacks that an adversary can make. Using symbolic execution over this model, we generate a set of mathematical constraints, where each constraint characterizes the set of secret values that lead to the same *sequence* of side-channel measurements. We then compute the optimal attack, i.e, the attack that yields maximum leakage over the secret, by solving an optimization problem over the computed constraints. We use information-theoretic concepts such as channel capacity and Shannon entropy to quantify the leakage over multiple runs in the attack, where the measurements over the side channels form the *observations* that an adversary can use to try to infer the secret. We also propose greedy heuristics that generate the attack by exploring a portion of the symbolic attack model in each step. We implemented the techniques in Symbolic PathFinder and applied them to Java programs encoding web services, string manipulations and cryptographic functions, demonstrating how to synthesize optimal side-channel attacks.**

***Index Terms*—Side-Channel Attacks; Quantitative Information Flow; Cryptography; Multi-run Security; Symbolic Execution; Satisfiability Modulo Theories; MaxSMT; Model Counting**

I. Introduction

Due to the widespread use of computers and other smart devices in every aspect of modern life, many software systems have access to secret information ranging from financial and medical records of individuals to trade secrets of companies and military secrets of states. Confidentiality of secret information is critical, however it is hard to achieve if an adversary is able to mount side-channel attacks [1], [2]. Side-channel attacks aim to recover secrets by observing non-functional aspects of program behavior such as execution time, network traffic or memory usage.

In this paper, we propose *symbolic analysis* techniques for detecting and measuring side-channel vulnerabilities and for synthesizing inputs that exploit the identified vulnerabilities. We consider side-channel attacks that span multiple runs and are *adaptive*; at each run, the adversary chooses a value for the public input, taking into account the outcomes of previous runs. This corresponds to the typical case of an attacker that tries to gradually uncover a secret that is constant across program runs, such as the secret key used in an encryption/decryption algorithm.

We start with a simple, generic *symbolic attack model* that encodes all the side-channel attacks that an adversary can make, up to some specified number of steps. We analyze this model using symbolic execution [3], a well known program analysis technique that computes program behaviors in terms of mathematical constraints over the specified symbolic inputs. The generated constraints describe partitions on the secret values, where each block in a partition contains the secrets that lead to the same *sequence* of side-channel measurements.

We compute the *optimal* attack, i.e. the attack that results in maximum leakage over the secret, by solving optimizations problems over the computed constraints. We use information-theoretic concepts such as channel capacity and Shannon entropy to quantify the information leakage over multiple runs in the attack, where the measurements over the side channels form the *observations* that an adversary can use to try to infer the secret. Solving the optimization problems produces a maximal assignment for the public inputs (“the attack”) over the set of constraints obtained with symbolic execution, yielding maximal leakage i.e. any other assignment produces less leakage.

The optimal attack reveals the most vulnerable behaviors of the program and the public (low) inputs that trigger those behaviors. This information can be used by the developers to understand the attacks and defend against them. For example, the attack trees that we generate automatically can be used as *attack signatures* to filter actual attacks during operation. Our work is also relevant in the context of the single-run attacker model [4], [5], [6], [7], [8], [9], [10], [11]. Most previous work addresses the computation of information leakage assuming the low input is fixed and do not address the problem of *generating* the low inputs that reveal the side channels. However, in most realistic scenarios different low inputs yield different leakages and thus the choice of the low inputs is important. Our work can be used to *automatically generate* low inputs that reveal side channels with the largest leakage.

We propose different approaches for synthesizing optimal attacks, where optimality is defined with respect to the information-theoretic measure that we use. In the first approach (MaxCC), we synthesize optimal attacks with respect to channel capacity using MaxSMT solving [12]. Channel capacity only takes into account the *number* of observations generated by the system. We also consider the harder problem of computing the inputs that maximize leakage with respect to entropy. This measure can give more precise estimates on the leakage but it is more difficult to compute, since it involves not only the number of observations that are made but also the *sizes* of the blocks in the induced partitions on the secret. We introduce a novel algorithm (MaxHMarco) for the precise computation of the optimal attack with respect to entropy, based on a solution of the Maximal Satisfiable Subsets problem [13]. The algorithm is general and works for any notion of entropy which is monotonic with respect to refinement ordering of partitions. We compare this algorithm with a third approach that we propose (MaxHNumeric) which synthesizes adaptive attacks again with respect to entropy, using parameterized model counting over the blocks in the partions on the secret followed by numeric optimization methods. The parameterized computations can be used for optimizing other measures such as guessability or Min entropy. MaxHNumeric can scale better with respect to the domain size but is not guaranteed to find an optimal attack. However, we found that in practice, MaxHNumeric computes attacks that are often close to optimal.

Since full exploration of the symbolic attack model can be computationally expensive, we propose greedy heuristics that generate the attack by exploring a portion of the symbolic attack model in each step. We show experimentally that the greedy heuristics generate optimal attacks for side channels with "segmented oracles", where the adversary is able to explore each segment of a secret independently. These side channels are typically found in password checking or file compression applications that use early termination optimization. Further, we are able to automatically generate optimal attacks for programs with side-channels that allow a binary search or a range search within the domain of the secret.

To deal with the imprecision that is inherent in the side-channel measurements, we also introduce a layer of *abstraction* in the cost model that we use for computing the side-channel measurements. The abstraction groups together the observations that may be indistinguishable to the adversary, resulting in a more realistic, coarser cost model and improving the scalability of the analysis, while enabling the discovery of real vulnerabilities.

We have implemented the proposed techniques in the Symbolic PathFinder (SPF) tool [14] and show their merits in the context of side-channel analysis for complex programs encoding web services, string manipulations and cryptographic functions, demonstrating how our techniques compute tight bounds on side-channel leakage and synthesize optimal side-channel attacks. Our implementation is general and can be easily implemented in other symbolic execution tools, for different languages.

II. Multi-Run Adaptive Attacks

Let $P(H, L)$ be a *deterministic* program, where H denotes the high input (secret) and L the low input (public). Similar to previous work [15], we assume that the attacker can make one side-channel observation at a time and that there are no errors in the measurements. We also assume that the attacker knows the implementation of P. These are strong assumptions that are justified since we are interested in computing the "strongest" possible attack.

In general, the attacker can not learn all the secret from only one round of observation. We are interested in computing the low values for estimating the maximum leakage after the attacker runs the program multiple times and makes multiple side-channel observations to *gradually* uncover information on the secret. One can try to infer this maximal leakage based on the computation of leakage on a single run, but this computation would not be accurate, since the attacker can *learn* from previous tries and will try to pick different low values that uncover different information in each round.

1) Attacker Model: The attacker can be defined as a (partial) function A that takes the history of observations as input and returns the low value l to be used in the next attack step.

We can model the interaction, up to k steps, between the attacker and the program as a system $S = (A, P, k, cost(\cdot))$, where the attacker A generates values of l for multiple executions of program P in order to determine the secret h. Parameter $cost(\cdot)$ determines the side-channel observations for program executions.

We assume that each path can give only one observable. Our work is done in the context of a project that targets side-channels for Java programs, where this assumption holds. Our work is also applicable to more general quantitative information flow analysis where the same assumption holds.

System S is defined as follows:

Procedure: A k-step adaptive attack $S = (A, P, k, cost(\cdot))$

```
1 $seq \leftarrow \emptyset$
2 for $i \leftarrow 1$ to $k$ do
3     $l \leftarrow A(seq)$
4     $o \leftarrow cost(P(h, l))$
5     $seq \leftarrow append(seq, o)$
```

The attacker is *adaptive* as it picks a new low value with each observation made. In contrast, a non-adaptive attacker would only pick one low value at each attack step, *regardless* of the observations made in previous runs, i.e. the attack function A would be a function of attack step i (and the low value would be $l \leftarrow A(i)$). Thus adaptive attacks can be more powerful, since the attacker can pick different low values at same step.

2) The Attacker's Knowledge: Suppose that the attacker observed o_1, o_2, .. o_k after picking values L_1, L_2 .. L_k and executing the program k times. The initial domain of the secret is D. At each step i, the attacker learns that the secret *leads* to o_i under L_i and revises the domain to contain those secrets that are consistent with this new observation. The attacker successfully reveals the secret when she can deduce the domain to consist of only one value. However, if after an attack step, the revised domain stays the same, the adversary does not learn new information with this low input.

In general, an adaptive attack can lead to different sequences of observations, depending on the secret value. Following [15] we say that two secrets H and H' are *indistinguishable* under

```
int secret;
int public_input;

if(secret >= public_input)
  ... perform some computation; //cost=1
else
 ... perform some other computation; // cost=2
```

Fig. 1. Illustrative Example

the attack A (written as $H \approx H'$) if for all observation sequences seq, $cost(P(H, A(seq))) = cost(P(H', A(seq)))$.

Proposition 1: The indistinguishability relation under attack A forms an equivalence relation on the secret values.

Proof: Reflexivity, symmetry and transitivity are easy to check because the program is deterministic. ■

Thus the attack A induces a *partition* on the secret values. Each block in the partition contains the secret values that lead to the same observations, under an attack A. Furthermore, the size of the partition is equal to the number of different k-sized observation sequences produced under A.

Given the system $S = (A, P, k, cost(\cdot))$ we can extend the classical definitions of Channel Capacity and Shannon entropy to reasoning about *sequences* of observations in S (henceforth called k-observables).

The channel capacity theorem [16], [17] states that the leakage for a program (in number of bits on the secret) is always less than or equal to the log of the number of possible distinct observations that an attacker can make. The result states in essence that leakage computation reduces to *counting* the number of different observable outputs for the program.

Thus, if running system S for a particular attack A results in $NkObs$ k-observables, the information leaked by P after k runs is:

$$\text{Information leaked after k runs} \leq CC^k(P) = \log_2(NkObs)$$

For deterministic systems, the Shannon entropy gives a measure of the leakage of the side-channel, corresponding also to the observation gain (on the secret) after an observation. We extend this result to multiple observations as follows. For each sequence of observations $o_i^k = \langle o_1, o_2, ..o_k \rangle$, let $p(o_i^k)$ denote the probability of observing o_i^k. Then the Shannon entropy is:

$$\mathcal{H}^k(P) = -\sum_{o_i^k} p(o_i^k)\log_2(p(o_i^k)) \quad (1)$$

Different attacks A lead to different leakage. The *most powerful attacker* will want to pick the low values that will leak the most information about the secret. In particular we are interested in *synthesizing* function A that maximizes Channel Capacity or Shannon entropy. Our work generalizes to other information theoretic measures such as computing the probability of guessing the secret or the Min entropy.

3) Example: We illustrate our approach on the example from Fig. 1. The program performs two kinds of operations (of costs 1 or 2) according to the branching condition in the code. Assume that the domain of the secret is $D = 1..6$. Consider a 2-step attack that picks low value 4 in the first step, low value 5 after seeing cost $\langle 1 \rangle$ and low value 3 after seeing cost $\langle 2 \rangle$, i.e. $A(\emptyset) = 4, A(\langle 1 \rangle) = 5, A(\langle 2 \rangle) = 3$. Suppose the attacker first observes $\langle 1 \rangle$. She can then deduce that the value of the secret is greater or equal than 4, thus narrowing down the possible values of the secret to $\{4, 5, 6\}$. Running the program again (on $A(\langle 1 \rangle) = 5$) and after observing $\langle 2 \rangle$, she can deduce that the secret is less than 5, narrowing down the possible secret values to $\{4\}$. Thus the attacker is able to fully recover the secret along this path. If on the other hand the second observation is $\langle 1 \rangle$, this means that the secret is greater or equal than 5, so the attacker is able to narrow down the secret to two values $\{5, 6\}$ etc.

4) Adaptive Attacks as Games: As mentioned We consider the case of adaptive attacks, where the attacker can take into account the outcomes of his previous queries, when deciding which low value to pick for the next query. The attacker's strategy is modelled as (partial) function A from sequence of observables (history) to low inputs (the next input). Note that the strategy is deterministic, since at each step the attacker decides on a unique next low value, according to its strategy.

We can see the above defined function as a strategy in a two players zero sum game (played between an attacker and a defender) defined as follows: The attacker moves are low inputs and an attacker strategy is a partial function A as defined above. The defender moves are observations: given a low input played by the attacker the defender plays the observable which is obtained by running the program on that low input. The (unique) defender strategy in this game is the function from low inputs to observables as prescribed by the program. A state of the game is a pair of strategies (one for the defender and one for the attacker). A payoff in this game is a function from a state of the game to real numbers.

Notice that the defender strategy is unique hence in defining the payoff we only need to consider the attacker strategy. Given an attacker strategy we can uniquely associate to that strategy a payoff given by the channel capacity or Shannon entropy defined in the previous section. Note that the indistinguishability relation under strategy A can be computed as the least upper bound computed over the indistinguishability relations induced by each low input defined by the strategy A; thus the payoff captures complete information for the strategy.

We seek to study the strategies in this game that have the highest payoff. As the defender strategy is unique such strategies only depend on the attacker and are the pure strategies with the highest payoff for the attacker or a mixture of such strategies.

Clearly these strategies are optimal, i.e. cannot be improved upon, with respect to the payoff function. Notice that optimality here is defined only with respect to a chosen payoff function and so it varies changing said function.

If we consider for example the channel capacity payoff then an optimal attacker strategy has a simple information theoretical semantics: faced with a set of possible secrets and and a number of partitions over this secret (each partition provided by a particular choice of a sequence of low inputs) the adversary chooses a maximal partition over the k runs because this is one providing maximal information in that number of runs. However that choice is only optimal up to k runs, i.e. if the game over k+1 runs we could find a partition, and hence a strategy improving wrt the same criteria.

III. Symbolic Execution for Attack Synthesis

We use symbolic execution to synthesize the attack that maximizes Channel Capacity and Shannon Entropy. Symbolic execution is a program analysis technique that executes a program on symbolic inputs, representing multiple concrete inputs. The result of the analysis is a set of symbolic paths each with a *path condition*, which is a conjunction of constraints over the symbolic inputs that characterizes all the concrete inputs that follow that path. To deal with loops and recursion a bound is put on the analysis depth.

We first create a *symbolic* model of the attack scenario described in the previous section. We model the secret using a symbolic variable. Furthermore, since we do not know in advance the low values that give the maximal leakage, we model them as fresh symbolic variables, as well. The resulting system, S_{sym} is described below:

Procedure: Symbolic model of a k-step adaptive attack: $S_{sym}(P, k, cost(\cdot))$

```
1 $seq \leftarrow \emptyset$
2 $h \leftarrow makeSymbolic(''h'')$
3 for $i \leftarrow 1$ to $k$ do
4     $l \leftarrow makeSymbolic(\mathcal{N}(seq))$
5     $o \leftarrow cost(P(h, l))$
6     $seq \leftarrow append(seq, o)$
```

The code is similar to the procedure S shown in Section II except that we use directive $makeSymbolic(name)$ to create high and low symbolic values with the specified name. Notably, at each iteration we create a symbolic value for low ($\mathcal{N}(seq)$), whose name is a *function* of observation sequence seq. Intuitively, this mimics the fact that after observing sequence seq, the attacker learns the information about h that is consistent with these observations, and chooses the next low value accordingly. However, instead of fixing an attack A a-priori and choosing concrete low values based on it, we leave the low values symbolic, encoding all the possible concrete values at each attack step.

Running symbolic execution on S_{sym} will generate a set of *symbolic paths* corresponding to the k invocations of program P. These paths represent all the possible attacks of an adaptive adversary up to k runs, since the symbolic values introduced in S_{sym} represent all the possible concrete values (of high and low). Furthermore, the symbolic paths generated with a symbolic execution of a system represent all the possible concrete paths through that system [3].

For simplicity we will assume that all the paths terminate within the prescribed bound. As this is not always the case in general, in practice we can use a notion of *confidence* (similar to [18]) that quantifies the impact of the execution bound on the quality of the analysis.

Each path π is a composition of paths $\pi_1; \pi_2..\pi_k$, where each π_i is a path in the i-th invocation of P. The *cost* of π is a k-observable, i.e., a sequence of k side-channel measurements made during the attack. Each path has a corresponding path condition, $PC^k(h, \bar{l})$, which in turn is a *conjunction* of k path conditions obtained from single invocations of P (as prescribed by S_{sym}). Here h denotes the symbolic value of the secret while $\bar{l}$ denotes a *tuple* of symbolic low values as created by S_{sym}. Note that there is a one-to-one correspondence between paths and path conditions. We write $cost(PC^k)$ to denote the k-observable for the corresponding path.

A value assignment for the symbolic low variables defines a concrete attack. In particular, let $\mathcal{V}$ be a function that assigns to each symbolic low variable a value from low input's domain. We synthesize A by defining, for each low variable of the form $\mathcal{N}(seq)$,

$$A(seq) = \mathcal{V}(\mathcal{N}(seq))$$

Different value assignments result in different attacks. For each concrete attack we can compute the channel capacity and Shannon entropy as follows. Let $\bar{L} = \langle \mathcal{V}(l_1), \mathcal{V}(l_2), ... \rangle$ denote a value assignment for a concrete attack. For each k-observable o_i^k we build a clause C_i:

$$C_i(h, \bar{L}) = (\bigvee_{cost(PC_j^k) = o_i^k} PC_j^k(h, \bar{L}))$$

Intuitively each clause characterizes all the secrets that are indistinguishable under the attack defined by $\bar{L}$. Let $\mathcal{C}$ be the set of all satisfiable clauses. The channel capacity is then:

$$CC^k(P) = \log_2(|\mathcal{C}|)$$

Further we compute the Shannon entropy using model counting over the clauses to compute the probabilities for each observation. For a uniform distribution over the secret, the probability of observing o_i^k is: $p(o_i^k) = \frac{\sharp(C_i(h, \bar{L}))}{\sharp D}$.

Here $\sharp(c)$ denotes the number of solutions, i.e., possible values satisfying the constraint c. This count can be computed with an off-the-shelf model-counting procedure such as Barvinok [19]. We use $\sharp D$ to denote the size of the secret domain D assumed to be (possible very large but) finite. Then leakage according to Shannon entropy is defined as follows:

$$\mathcal{H}^k(P) = -\sum_{i=1,m} \frac{\sharp(C_i(h, \bar{L}))}{\sharp D} \log_2(\frac{\sharp(C_i(h, \bar{L}))}{\sharp D})$$

We are interested in finding the assignment $\bar{L}$ of low variables that yield the maximal leakage according to the two formulas above. Intuitively this value assignment will allow us to compute *bounds* on the information leakage that an attacker can achieve. We reduce the problem of finding this value assignment to optimization problems over the set of clauses, as described later in this section.

1) Example: Consider our running example. The result of symbolically executing S_{sym} for the program is a set of symbolic paths which can be organized in a tree as shown in Figure 2 (for $k = 3$). In the tree, nodes depicted with bold rectangles represent "attacker moves", i.e., choosing low values based on the history of observations. Nodes depicted with light rectangles represent "system responses", i.e., the side-channel measurements made along program runs. We also depict the intermediate path conditions computed for the different observations. The path conditions encode the constraints on the secret that the attacker is *learning* with each observation, while attempting to narrow down the values of the secret. The path constraints on the leaves represent the knowledge that the attacker has learned after k steps.

The tree is interpreted as follows. In a first step the attacker chooses symbolic value l for low and runs the program once obtaining observations $\langle 1\rangle$ and $\langle 2\rangle$, with path conditions $h \geq l$ and $h < l$ respectively. The attacker, then runs the program a second time, using a new symbolic value l_1 if the observed cost is $\langle 1\rangle$ and a new symbolic value l_2 if the observed cost is $\langle 2\rangle$. This second execution results in more constraints on the secret, that help the attacker narrow down its values. For example, if the observed cost sequence is $\langle 1, 2\rangle$, one can determine that the secret satisfies the constraint $h \geq l \wedge h < l_1$.

Note that the tree encodes multiple concrete attacks all at once. For example, consider an attack that in the first run picks low value 4, while in the second run it picks 5 after observing $\langle 1\rangle$ and 3 after $\langle 2\rangle$. This corresponds to variable assignment: $l = 4, l_1 = 5, l_2 = 3$. After $\langle 1, 2\rangle$ the attacker has learned that $h \geq 4 \wedge h < 5$ which narrows down the possible values of h to only one value (4). Consider now a different variable assignment: $l = 4, l_1 = 6, l_2 = 3$. After $\langle 1, 2\rangle$ the attacker has learned that $h \geq 4 \wedge h < 6$ which narrows down the possible values of h to two (4 and 5) etc.

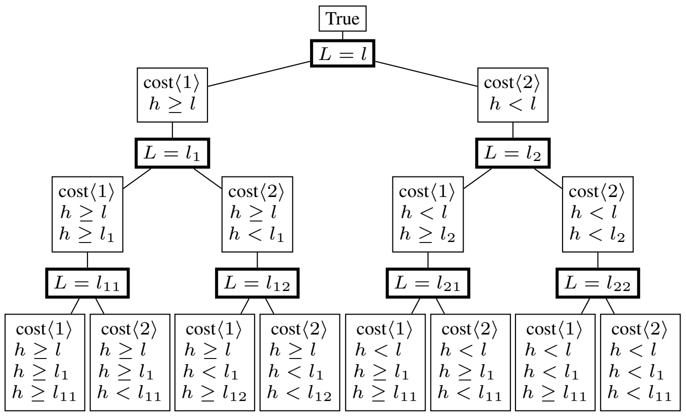
True
$L = l$
cost⟨1⟩ $h \geq l$
cost⟨2⟩ $h < l$
$L = l_1$
$L = l_2$
cost⟨1⟩ $h \geq l$ $h \geq l_1$
cost⟨2⟩ $h \geq l$ $h < l_1$
cost⟨1⟩ $h < l$ $h \geq l_2$
cost⟨2⟩ $h < l$ $h < l_2$
$L = l_{11}$
$L = l_{12}$
$L = l_{21}$
$L = l_{22}$
cost⟨1⟩ $h \geq l$ $h \geq l_1$ $h \geq l_{11}$
cost⟨2⟩ $h \geq l$ $h \geq l_1$ $h < l_{11}$
cost⟨1⟩ $h \geq l$ $h < l_1$ $h \geq l_{12}$
cost⟨2⟩ $h \geq l$ $h < l_1$ $h < l_{12}$
cost⟨1⟩ $h < l$ $h \geq l_1$ $h \geq l_{11}$
cost⟨2⟩ $h < l$ $h \geq l_1$ $h < l_{11}$
cost⟨1⟩ $h < l$ $h < l_1$ $h \geq l_{11}$
cost⟨2⟩ $h < l$ $h < l_1$ $h < l_{11}$

Fig. 2. Symbolic tree for running example.

A. Maximizing Channel Capacity

We compute the optimal strategy with respect to channel capacity using MaxSMT solving, by extending the result from [20] which, however, only applied to non-adaptive attacks.

MaxSMT [12] is an extension of SMT (satisfiability modulo theories) solving to optimization: given a weighted first-order formula composed of a set of clauses, each with a weight (positive or infinity), MaxSMT finds the assignment that minimizes the sum of the weights of the falsified clauses, or alternatively maximizes the sum of satisfied clauses.

In our setting we consider a set $\mathcal{C} = \{C_1, C_2 \ldots C_n\}$ of clauses, where each C_i has the weight 1. The MaxSMT problem is then to find a subset $\mathcal{M} \subseteq C$ with largest cardinality, such that $\mathcal{M}$ is satisfiable.

The approach is illustrated in Algorithm 1. Procedure ComputeConstraints builds a set $\mathcal{C}$ of clauses, where, as before, each clause $C_i(h, \bar{l})$ is a disjunction of the path conditions leading to the same k-observable. Note, however, that the values of low are left symbolic. This set is processed by procedure MaxCC as follows. We transform $C_i(h, \bar{l})$ into $C_i(h_i, \bar{l})$ by renaming h with fresh h_i in each clause C_i, respectively. The intuition is the same as in [20]: the clauses are renamed to define constraints on low variables, while the high variables are left unconstrained and the goal is to find the low input value that leads to the maximum number of observations for *any value* of the secret.

```
Algorithm 1: AdaptiveMaxLeakCC($P, k, cost(\cdot)$)
1 $\mathcal{C} \leftarrow ComputeConstraints(P, k, cost(\cdot))$
2 $(w, \bar{L}) \leftarrow MaxCC(\mathcal{C})$
3 return $(log_2 w, \bar{L})$
```

```
Procedure: ComputeConstraints($P, k, cost(\cdot)$)
1 $\mathcal{O} \leftarrow \emptyset, \mathcal{C} \leftarrow \emptyset$
2 $\mathcal{PC} \leftarrow SymEx(S_{sym}(P, k, cost(\cdot)))$
3 foreach $PC_i^k(h, \bar{l}) \in \mathcal{PC}$ do
4     $\mathcal{O} \leftarrow \mathcal{O} \cup \{cost^k(PC_i^k(h, \bar{l}))\}$
5 foreach $o_i^k \in \mathcal{O}$ do
6     $C_i(h, \bar{l}) \leftarrow \bigvee_{cost(PC_j^k) = o_i^k} PC_j^k(h, \bar{l})$
7     $\mathcal{C} \leftarrow \mathcal{C} \cup \{C_i(h, \bar{l})\}$
8 return $\mathcal{C}$
```

```
Procedure: MaxCC($\mathcal{C}$): Maximizing Channel Capacity
1 $\mathcal{C} \leftarrow Rename(\mathcal{C})$
2 $(w, \bar{L}) \leftarrow MaxSMT(\mathcal{C})$
3 return $(w, \bar{L})$
```

Applying MaxSMT to the set of renamed clauses will yield the *maximal* number of clauses that are together satisfiable, and thus yield the maximum number of observations possible (up to k), giving maximum leakage in terms of channel capacity. Further, MaxSMT gives a *solution*, i.e., an assignment $\bar{L}$ to symbolic variables $\bar{l}$ that satisfies the maximum satisfiable clauses, meaning that it induces the partitioning on the secret with the maximum number of equivalence indistinguishability classes, and thus it defines the best k-step attack.

Proposition 2: Algorithm AdaptiveMaxLeakCC computes the k-step adaptive attack that is optimal w.r.t. Channel Capacity.

1) Example: As an illustration, consider again the running example. The analysis (up to k=3) yields 8 path conditions, corresponding to cost sequences: $\langle 1,1,1\rangle$, $\langle 1,1,2\rangle$, $\langle 1,2,1\rangle$, $\langle 1,2,2\rangle$, $\langle 2,1,1\rangle$, $\langle 2,1,2\rangle$, $\langle 2,2,1\rangle$, $\langle 2,2,2\rangle$. Each symbolic path yields different cost, thus each path condition corresponds to a clause, giving the following clauses (after renaming):

$$\begin{array}{ll}
h_1 \geq l \wedge h_1 \geq l_1 \wedge h_1 \geq l_{11} & h_2 \geq l \wedge h_2 \geq l_1 \wedge h_2 < l_{11} \\
h_3 \geq l \wedge h_3 < l_1 \wedge h_3 \geq l_{12} & h_4 \geq l \wedge h_4 < l_1 \wedge h_4 < l_{12} \\
h_5 < l \wedge h_5 \geq l_2 \wedge h_5 \geq l_{21} & h_6 < l \wedge h_6 \geq l_2 \wedge h_6 < l_{21} \\
h_7 < l \wedge h_7 < l_2 \wedge h_7 \geq l_{22} & h_8 < l \wedge h_8 < l_2 \wedge h_8 < l_{22}
\end{array}$$

Solving with MaxSMT gives that the maximum number of satisfiable clauses, corresponding to maximum number of observables after 3 steps, is 6, which is equal to the domain of the secret. Thus, no matter what the value of the secret is (in domain 1..6) an attacker can guess it in maximum 3 steps. Furthermore, MaxSMT provides a satisfying assignment to the values in $\bar{l}$: $l = 3, l_1 = 5, l_2 = 2, l_{11} = 6, l_{12} = 4$ defining an attack as illustrated in Figure 3. The leaves in the tree define the partition on the secret induced by this attack. All the blocks in the partition have size 1, confirming that the attacker can always guess the secret for this example. Note that the partitions on the right sub-tree in the figure already have size 1 after two steps. Thus, a third attack step is not necessary – in our implementation we exploit such situations to perform *pruning* of the attack tree, as explained later in this section.

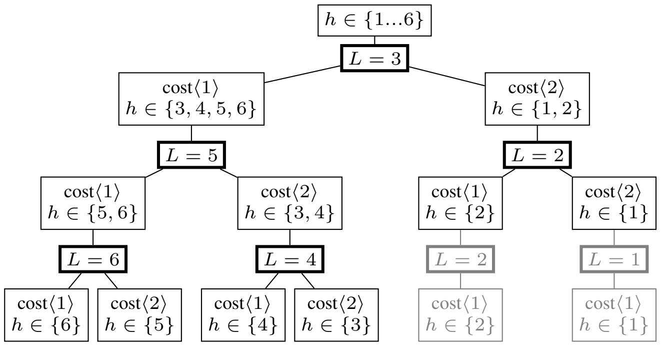

Fig. 3. Attack tree computed with MaxSMT.

B. Maximizing Shannon Entropy

Computing the low inputs (i.e., the attack) that maximize the number of observations does not necessarily lead to the optimal attack with respect to Shannon entropy. We propose alternative strategies that aim to maximize Shannon entropy instead of simply the number of observations.

As described earlier, an attack consists of an assignment of concrete values to symbolic low inputs, $\bar{L} = \langle \mathcal{V}(l_1), \mathcal{V}(l_2), ...\rangle$. Our goal is to choose $\bar{L}$ which maximizes the Shannon entropy given that choice of $\bar{L}$, which we denote $\mathcal{H}^k(P|\bar{L})$. Maximizing this entropy thereby maximizes the expected information leakage after k steps. To achieve this we developed two different methods, MaxHMarco and MaxHNumeric, which are both phrased as combinatorial optimization problems over $\bar{l}$ with objective function $\mathcal{H}^k(P|\bar{l})$. These two methods are complementary: MaxHMarco is guaranteed to return the partition with highest entropy but it is sensitive to the size of the input domain; MaxHNumeric uses numeric optimization methods that are approximate and therefore can not provide full guarantees but they can potentially scale to larger input domains.

Algorithm 2: AdaptiveMaxLeakH$(P, k, cost(\cdot))$

```
1 $\mathcal{C} \leftarrow ComputeConstraints(P, k, cost(\cdot))$
2 $\bar{L} \leftarrow MaxH(\mathcal{C}, true)$
3 **return** $(\mathcal{H}^k(P|\bar{L}), \bar{L})$
```

Algorithm 2, AdaptiveMaxLeakH, outlines our approach. We first use symbolic execution to compute the set of clauses $\mathcal{C}$ which partitions the input. We then compute the value of $\bar{L}$ which maximizes the entropy by setting the function MaxH to be either MaxHMarco or MaxHNumeric, and finally return the maximum entropy value, which gives the information leakage. The two methods are detailed in the subsequent sections.

C. Entropy Maximization Via Maximal Satisfiable Subsets

The MaxSMT solution described in the previous section represents only one maximal partition on the secret which may not necessarily lead to maximum entropy. We therefore propose to compute *all* the maximal partitions on the secret, from which we choose the one that has maximal entropy.

To this end we use and extend the MARCO algorithm [13], which solves a generalization of MaxSMT, namely the problem of finding *all* Maximal Satisfiable Subsets (MSSs) of clauses that are together satisfiable.

Let $\mathcal{C}$ be a set of clauses of the form $C(h, \bar{l})$ as before, where each clause has the weight 1. Set $M \subseteq \mathcal{C}$ is a *maximal satisfiable subset* (MSS) if (the conjunction of all clauses in) M is satisfiable and $\forall C \in \mathcal{C} \setminus M : M \cup \{C\}$ is unsatisfiable. Note that any solution to the MaxSMT problem is an MSS, while some MSSs may have smaller cardinality than the maximum size.

Each MSS contains a set of clauses $\{C_1(h, \bar{l}), C_2(h, \bar{l}), ..\}$ which are together satisfiable. Furthermore, each variable assignment $\bar{L}$ over these clauses defines a partition on the secret, where the secret values in each block satisfy a clause $C_i(h, \bar{L})$. We therefore generate, within each MSS, all the partitions on the secret by repeatedly finding satisfiable assignment for low values, and adding a blocking clause to find new partitions. Note that this is not necessarily the same as enumerating all the possible low values. We only search for different partitions within the same maximal satisfiable subset.

Procedure MaxHMarco describes our approach. We use the Marco algorithm to discover the set $\mathcal{M}$ of all MSSs; each MSS is a maximal satisfiable subset of set of (renamed) clauses. Formula $\varphi(\bar{l})$ is the conjunction of all the clauses in the MSS

Procedure: MaxHMarco$(\mathcal{C}, D)$: Maximizing Entropy

```
1  $\mathcal{C} \leftarrow Rename(\mathcal{C})$
2  $\mathcal{M} \leftarrow Marco(C)$
3  $leak_{max} \leftarrow 0$
4  foreach $MSS \in \mathcal{M}$ do
5      $\varphi(\bar{l}) \leftarrow true$
6      foreach $C_i(h_i, \bar{l}) \in MSS$ do
7          $\varphi(\bar{l}) \leftarrow \varphi(\bar{l}) \wedge C_i(h_i, \bar{l})$
8      while $isSAT(\varphi(\bar{l}))$ do
9          $\bar{L} \leftarrow getModel(\varphi(\bar{l}))$
10         foreach $C_i(h_i, \bar{l}) \in MSS$ do
11             $p(o_i^k) \leftarrow \sharp(C_i(h_i, \bar{L}))/\sharp D$
12         $leak \leftarrow -\sum_{o_i^k} p(o_i^k)\log_2(p(o_i^k))$
13         if $leak_{max} < leak$ then
14             $leak_{max} \leftarrow leak$
15             $\bar{L}_{max} \leftarrow \bar{L}$
16         $\varphi(\bar{l}) \leftarrow \varphi(\bar{l}) \wedge \neg\varphi(\bar{L})$
17 return $L_{max}$
```

(for clarity we omit the h values). Each concrete low input $\bar{L}$ defines a partition on the secret h that satisfies the clauses in MMS, i.e., $\varphi(\bar{L})$ is satisfiable. We force the SMT solver to find different partitions by adding a blocking clause (line 13):

$$\varphi(\bar{l}) \leftarrow \varphi(\bar{l}) \wedge \neg\varphi(\bar{L})$$

We discover all the partitions of the secret associated with this MSS, when φ becomes unsatisfiable. Since the domain of $\bar{l}$ is finite, this is guaranteed to terminate.

Proposition 3: For any distribution on the secret the attack of maximal entropy is given by the entropy of a partition in the set $\mathcal{M}$.

Proof: By construction $\mathcal{M}$ contains all MSSs. This means that any other sub-set $S \subseteq \mathcal{C}$ such that $S \notin \mathcal{M}$ is included by an MSS, i.e., $\exists M \in \mathcal{M}$ such that $S \subseteq M$. It follows that for any low value L, the partition induced by M is *finer* than the partition induced by S (here we consider the refinement order on partitions of the lattice of information from [21], i.e., each block in the partition M is included in a block in the partition S). Thus, according to [21], it follows that the entropy induced by M is greater than the one induced by S, because the entropy is a monotonic function w.r.t. the lattice of information ordering (for any distribution on the secret). Thus, we are guaranteed to find the partition with maximal entropy by only searching through the MSSs. ■

MaxHMarco can be used with minimal modification for computing the optimal attack with respect to other leakage measures, e.g. to use for guessability we only need to replace
$leak \leftarrow -\sum_{o_i^k} p(o_i^k)\log_2(p(o_i^k))$ with
$leak \leftarrow \sum_{b_i}(b_i/\sharp D)(b_i+1)/2$ where $b_i = \sharp(C_i(h_i, \bar{L}))$
and replace the test **If** $leak_{max} < leak$ with **If** $leak_{max} > leak$. Here guessability is the average number of guesses required to guess the secret [15]. Notice that proposition 3 holds for guessability too once we make the above changes to MaxHMarco. In fact guessability is anti-monotonic w.r.t. the refinement order on partitions of the lattice of information [21] and as for guessability the test $leak_{max} < leak$ in MaxHMarco is inverted, monotonicity in the proposition is replaced by anti-monotonicity. Hence for guessability too we can restrict the search for the optimal attack within the set $\mathcal{M}$.

1) Example: Let us discuss again our running example (see Fig. 1). Consider for simplicity a single attack step (i.e., k=1). Performing symbolic execution on the code where `secret` has symbolic value h yields clause set $\mathcal{C} = h \geq l, h < l$. Running the Marco algorithm yields only one maximal satisfiable subset, $\mathcal{C}$ itself. Then $\phi(l)$ is $h_1 \geq l \wedge h_2 < l$. Suppose solving ϕ gives solution $l = 2$ (we ignore the solution for the high values). This solution induces the following partition on the secret: $\{1\}\{2, 3, 4, 5, 6\}$. Adding the blocking clause results in a new formula: $\phi(l) :: h_1 \geq l \wedge h_2 < l \wedge \neg(h_1 \geq 2 \wedge h_2 < 2)$. Solving this formula will yield a new solution (say $l = 3$) that is guaranteed to be different than the previous one (due to the blocking clause). This new solution yields a different partition on the secret: {1,2}{3,4,5,6}. Adding a new blocking clause for this solution will force the solver to generate a new partition on the secret and so on. Thus we can find all the secret partitions within the MSS. From these partitions, we select the partition with the highest entropy. For this example this corresponds to $l = 4$. Intuitively this is the *best* attack step, since $l = 4$ partitions the secret in *balanced* blocks of (almost) equal size.

D. Entropy Maximization Via Numeric Optimization

Our second approach generates a symbolic entropy function and attempts to directly maximize that function using numeric techniques. This method relies on parameterizing observation sequence probabilities by the choice of low input values, computed via model counting.

A model counting function for $C_i(h, \bar{l})$ is a function $F_i(\bar{l})$ that computes the number of possible secrets h that satisfy C_i, given a choice of $\bar{l}$. For instance, recall the running example from Fig. 1 with a secret domain $1 \leq h \leq 6$ and suppose we are interested in an attack for 2 steps. The adversary will input an initial guess l, and then input l_1 or l_2 depending on if cost $\langle 1 \rangle$ or cost $\langle 2 \rangle$ is observed. Then there are 4 possible constraints over the vector $\bar{l} = \langle l, l_1, l_2 \rangle$ corresponding to the leaves of the symbolic attack tree.

$$C_1 = h < l \wedge h < l_1 \qquad C_2 = h < l \wedge h \geq l_1$$
$$C_3 = h \geq l \wedge h < l_2 \qquad C_4 = h \geq l \wedge h \geq l_2$$

Consider the number of secrets h that are consistent with C_1 for a given choice of $\bar{l}$ and the domain of h. If both $l > 6$ and $l_1 > 6$ then h can take on any value in the domain and there are 6 solutions. If $1 \leq l \leq 6 \wedge l \leq l_1$ then there are exactly $l - 1$ values of h that satisfy C_1. Symmetrically, if $1 \leq l_1 \leq 6 \wedge l_1 < l$ then there are $l_1 - 1$ possible values for h. Otherwise, C_1 has no solutions. We can write a counting function for this and the 3 remaining constraints as piecewise functions (where it is assumed that if none of the piecewise conditions apply then the function is 0.)

$$F_1(\bar{l}) = \begin{cases} 6 & : l > 6 \wedge l_1 > 6 \\ l-1 & : 1 \leq l \leq 6 \wedge l \leq l_1 \\ l_1 - 1 & : 1 \leq l_1 \leq 6 \wedge l_1 < l \end{cases}$$

$$F_2(\bar{l}) = \begin{cases} 6 & : l_1 < 1 \wedge 6 < l \\ l-l_1 & : 1 \leq l_1 \leq l \leq 6 \\ l-1 & : l_1 < 1 \leq l \leq 6 \\ 7-l_1 & : 1 \leq l_1 \leq 6 < l \end{cases}$$

$$F_3(\bar{l}) = \begin{cases} 6 & : l < 1 \wedge 6 < l_2 \\ l_2-l & : 1 \leq l \leq l_2 \leq 6 \\ l_2-1 & : l < 1 \leq l_2 \leq 6 \\ 7-l & : 1 \leq l \leq 6 < l_2 \end{cases}$$

$$F_4(\bar{l}) = \begin{cases} 6 & : l < 1 \wedge l_2 < 1 \\ 7-l & : 1 \leq l \leq 6 \wedge l_2 < l \\ 7-l_2 & : 1 \leq l_2 \leq 6 \wedge l \leq l_2 \end{cases}$$

We use the parameterized model counter Barvinok [19] to automatically produce each $F_i(\bar{l})$. Barvinok performs parameterized model counting by representing a constraint C on variables $\bar{l}$ and h as a symbolic polytope $Q \subseteq \mathbb{R}^n$. Barvinok's algorithm generates a multivariate piecewise polynomial F such that $F(\bar{l})$ evaluates to the number of assignments of integer values to h that lie in the interior of Q.

Using each $F_i(\bar{l})$ we compute the probability of an observation sequence given the values of the low inputs as $p(o_i^k|\bar{l}) = F_i(\bar{l})/\#D$. We then plug these symbolic probability functions into Equation 1:

$$\mathcal{H}^k(P|\bar{l}) = -\sum_{i=1}^{m} \frac{F_i(\bar{l})}{\#D} \log_2 \frac{F_i(\bar{l})}{\#D}$$

Then, the attack synthesis can be stated as a non-linear objective function maximization problem, defined by $\bar{L} = \arg\max_{\bar{l}} \mathcal{H}^k(P|\bar{l})$. We leverage existing non-linear optimization routines to approximate $\bar{L}$. In our implementation we used MATHEMATICA's NMAXIMIZE. Our overall strategy generation algorithm using numeric entropy maximization is as follows:

Procedure: MaxHNumeric$(\mathcal{C}, D)$: Maximizing Entropy

```
1 foreach $C_i \in \mathcal{C}$ do
2     $F_i(\bar{l}) \leftarrow Barvinok(C_i, \bar{l}, h)$
3     $p(o_i^k|\bar{l}) \leftarrow F_i(\bar{l})/\#D$
4 $\mathcal{H}^k(P|\bar{l}) \leftarrow -\sum_{i=1}^{m} p(o_i^k|\bar{l}) \log_2(p(o_i^k|\bar{l}))$
  $\bar{L} = NMaximize(\mathcal{H}^k(P|\bar{l}))$
5 return $\bar{L}$
```

For the example from Figure 1, maximizing $\mathcal{H}(P|\bar{l})$ results in the assignment $\bar{L} = \langle 4, 2, 5 \rangle$ for symbolic inputs $\langle l, l_1, l_2 \rangle$. This corresponds to the first two steps of an adaptive timing side channel binary search attack.

Note that method MaxHNumeric can also be used, with minimal modifications, for computing an attack with respect to other measures. Once we have the (parameterized) probability computations (line 3) we can plug them in the formulas for, e.g., guessability or Min entropy, and apply numeric optimizations to maximize those measures.

E. Greedy Maximization

Generating the symbolic attack tree fully up to depth k will generate up to m^k knowledge states, where m is the number of observables. Thus, the maximization methods presented so far would involve over m^k low variables. Rather than perform the full exploration up to a given depth, we suggest a d-greedy approach, in which the attack is computed in phases of size d and the l-variables are solved to maximize channel capacity or entropy for each phase. This reduces the problem to solving $\frac{k}{d}$ maximization problems of size m^d, with the trade off of (possibly) yielding a suboptimal solution. Note that in the case of a binary search oracle side channel, as in the example, the 1-greedy solution using Shannon entropy happens to be the optimal solution, requiring solving k optimizations problems each with 1 free parameter, rather than 1 optimization problem with 2^k parameters. In general a greedy solution can be arbitrarily suboptimal [15].

F. Optimizations

Procedure ComputeConstraints$(P, k, cost(\cdot))$ is used to build a set of clauses, each one corresponding to a k-observable. The simplest and most intuitive implementation is to run Symbolic Execution on the system S_{sym}. However, this implementation can be optimized by running Symbolic Execution on only one copy of the program to obtain a set of path conditions. We then obtain the clauses corresponding to k-observables by systematically combining these path conditions (with appropriate renaming). This optimization reduces the overhead of symbolically executing the program multiple times. Further, for our greedy techniques we implemented early pruning to not expand attack steps for partition blocks that already have size 1, since no new information can be inferred for those blocks.

IV. Implementation and Experiments

We implemented the proposed techniques in the Symbolic PathFinder (SPF) [14] symbolic execution tool. SPF implements a custom JVM which symbolically executes Java bytecode instructions. We also provide a graphical display of the generated attacks. We use Barvinok [19] for (parameterized) model counting (for linear constraints). For numeric maximization we use MATHEMATICA's NMAXIMIZE function [22] configured to use *Differential Evolution* [23] and set to use between 100 and 250 iterations to balance running time and convergence.

1) Cost Models: Our work is done in the context of a project that specifically addresses side-channels that are related to time and space consumption in Java programs. To this end we implemented SPF listeners to monitor the

Case Study	DOMAIN	Steps	Full			1-greedy		
			maxObs	CC	time	maxObs	CC	time
Illustrative Example in Fig. 1	10	4	10	3.322	3.060	8	3	2.330
	200	8	-	-	-	64	6	10.026
		16	-	-	-	200	7.644	27.225
	300	9	-	-	-	83	6.375	11.561
		21	-	-	-	300	8.229	36.827
	400	9	-	-	-	97	6.600	13.143
		20	-	-	-	400	8.644	49.103
	500	9	-	-	-	71	6.150	10.049
		25	-	-	-	500	8.966	1m1.076
	10^6	10	1024	10	14.578	461	8.849	1m0.702
		11	2048	11	2m2.680	752	9.555	1m40.382
		12	4096	12	19m32.370	1199	10.228	2m39.689
		13	-	-	-	1903	10.894	4m15.845
java.util.Arrays.equals	256^3	4	35	5.129	15.717	35	5.129	8.668
		5	56	5.807	11m40.356	56	5.807	13.943
		6	-	-	-	84	6.392	20.981
	256^4	4	70	6.129	2m32.284	70	6.129	18.153
		5	-	-	-	126	6.977	32.997
CRIME	50^2	2	6	2.585	1.818	6	2.585	5.694
		3	10	3.322	8.533	10	3.322	9.301
		4	15	3.907	1m12.912	15	3.907	12.433
		5	21	4.392	15m5.197	21	4.392	15.783
		10	-	-	-	66	6.044	56.212
		20	-	-	-	231	7.852	5m29.748
		40	-	-	-	861	9.750	45m57.736
	50^3	2	10	3.322	1m11.818	10	3.322	9m18.541
		3	-	-	-	20	4.321	19m4.114
		4	-	-	-	35	5.129	29m30.333
LawDB	100 - 2	2	4	2	2.750	4	2	2.441
		3	8	3	18.915	7	2.807	3.244
		4	16	4	3m8.783	11	3.459	4.160
		5	-	-	-	17	4.087	5.342
		17	-	-	-	98	6.615	21.010
	10^6 - 2	2	4	2	2.762	4	2	2.579
		3	8	3	19.493	8	3	3.866
		4	16	4	3m21.688	16	4	5.844
		5	-	-	-	32	5	9.212
		17	-	-	-	6070	12.567	18m31.246

Fig. 4. Results for MaxCC (full exploration and 1-greedy).

bytecode instructions executed by the program, and to perform the analysis of side-channels related to time and space consumption. For timing channels we can compute the execution time of each (symbolic) path by assigning a time unit to each instruction and aggregating the cost. To obtain a more realistic cost model, we can also perform statistical measurements of the execution time of the program ran on a reference hardware platform, as driven by the tests that satisfy the corresponding path conditions. For space-related channels we monitor network and file communication, by providing models for network and file interactions and computing the number of bytes written to an output stream or file via methods `write` of `java.io.OutputStream` and `java.io.FileOutputStream` respectively. Analysis of other types of side channels can be implemented easily, e.g. one can monitor the memory allocated inside SPF's custom JVM to measure memory consumption. Presumably one can also implement cache side channels using the same JVM and monitoring for sequences of hits and misses.

We also developed an abstraction layer that groups together the costs that have very close values, as in practice they would be indistinguishable to an adversary. Let o_{min} and o_{max} be the minimum and maximum values of the costs observed along the paths in one run. We divide the range into n intervals from 0^{th} to $(n-1)^{th}$, where n is a user supplied parameter. Each interval has equal size, $d = \frac{o_{max}-o_{min}}{n}$. We then map all the costs *obs* such that $o_{min} + i \times d \leq obs < o_{min} + (i+1) \times d$ to the same interval i (o_{max} belongs to the $(n-1)^{th}$ interval). These intervals form the *abstractions* of the concrete costs and they are used in the analysis. In practice this abstraction can be

Case Study	MODULO	DOMAIN	Steps	Full			1-greedy		
				maxObs	CC	time	maxObs	CC	time
ModPow [20]	1717	2^3-1	2	7	2.807	7.679	7	2.807	3.572
		2^4-1	2	15	3.906	1m46.236	14	3.807	14.199
			3				15	3.906	28.350
		2^5-1	2	-	-	-	25	4.644	1m6.624
			3	-	-	-	30	4.906	5m36.462
			4	-	-	-	31	4.954	8m5.695
		2^6-1	2	-	-	-	34	5.087	9m47.837
			3	-	-	-	53	5.728	37m31.318
			4	-	-	-	-	-	-
	834443	2^3-1	2	7	2.807	10.704	7	2.807	3.624
		2^4-1	2	15	3.906	8m67.264	14	3.807	14.292
			3				15	3.906	30.856
		2^5-1	2	-	-	-	25	4.644	1m26.862
			3	-	-	-	30	4.906	3m7.863
			4	-	-	-	31	4.954	3m49.175
		2^6-1	2	-	-	-	46	5.524	4m42.696
			3	-	-	-	62	5.954	25m9.132
			4	-	-	-	63	5.977	35m53.526
	1964903306	2^3-1	2	7	2.807	13.124	7	2.807	3.269
		2^4-1	2	-	-	-	14	3.807	17.063
			3	-	-	-	15	3.906	27.107
		2^5-1	2	-	-	-	25	4.644	2m24.405
			3	-	-	-	30	4.906	4m1.789
			4	-	-	-	31	4.954	4m27.231

Fig. 5. Results for MaxCC on ModPow [20].

used to find optimal attacks, while also providing the benefit of greater scalability (since the number of observations and hence of clauses can be reduced significantly).

2) Experiments: There are three main techniques (for a depth k): (1) MaxCC, (2) MaxHMarco, and (3) MaxHNumeric, each with two variants: (a) full exploration and (b) 1-greedy approach. We evaluated both MaxCC (1a) and (1b) and observed that they generate effective attacks in reasonable time which we detail in the coming sections. For MaxHMarco and MaxHNumeric, we find that due to the complexity of composed constraints, (2a) and (3a) are not feasible in practice. In addition, (2b) and (3b) give optimal or near-optimal attacks. Thus we evaluate these four variants (1a, 1b, 2b, 3b) with the goal of assessing if they can automatically synthesize attacks and compute leakage for complex, realistic applications.

We analyzed the following programs: the standard Java API `java.util.Arrays.equals`, CRIME – an implementation of string compression that uses the Lempel-Ziv (LZ77) [24] algorithm, LawDB – a complex network service application[1], and ModPow – an implementation of modular exponentiation [20].

The results of the experiments for MaxCC (1a and 1b) are shown in Fig. 4 and Fig. 5, while the results of the experiments for MaxHMarco and MaxHNumeric (2b and 3b) are shown in Fig. 6. In the tables, `DOMAIN` is the number of possible values of the secret. The tables show the number of attack steps, the maximum number of observables, maxObs, the leakage and the analysis time (in seconds). A '-' indicates timeout (of 1h). All the experiments were run on a standard MacBook Pro. We first give a high level description of the discovered vulnerabilities and then we describe in more detail the results displayed in the tables.

[1]This program was provided to us by DARPA.

3) Vulnerabilities: Our techniques discovered a timing channel in `java.util.Arrays.equals` which is due to the early termination optimization. The method takes two arrays, compares the elements one by one and returns as soon as the elements are found to be different. When used to compare a secret and a public input, `java.util.Array.equals(low, high)`, an adversary can determine how many elements in the input array are matched with the secret, by measuring the execution time. She is then able to make repeated guesses on the next un-matched element until it is matched (the execution time is longer). An optimal attack is linear in the size of the secret, as the adversary can use the channel to guess each element one by one (as opposed to exponential for a simple brute force attack). This is an instance of a *segmented oracle* side channel which is studied in [25]. Our techniques computed the optimal attack automatically. We note that the use of `java.util.Arrays.equals` in Google's Keyczar library [26] led to a vulnerability that could allow an adversary to forge signatures for data that was "signed" with the default SHA-1 HMAC algorithm [27].

CRIME is inspired by the "Compression Ratio Info-leak Made Easy" attack [28]. We analyzed procedure: `compress(high.concat(low))`. Our techniques found a space side channel: the size of the compressed string leaks

Case Study	**DOMAIN**	**Steps**	**MaxHNumeric**		**MaxHMarco**		
			Leakage (bits)	time	maxObs	Leakage (bits)	time
Illustrative Example in Fig. 1	200	8	7.207	44.876	200	7.644	2m17.120
	300	9	7.560	69.383	300	8.229	3m52.363
	400	9	7.743	1m55.212	400	8.644	5m41.539
	500	9	7.800	1m24.068	500	8.966	7m36.105
	10^6	10	8.172	3m15.000	-	-	-
		11	8.303	4m55.088	-	-	-
		12	8.357	7m12.280	-	-	-
		13	8.371	9m34.512	-	-	-
		14	8.376	12m20.844	-	-	-
LawDB	100 - 2	2	1.999	2.552	4	1.999	1m5.234
		3	2.999	4.688	8	2.999	1m33.656
		4	3.998	10.284	16	3.998	1m49.308
		5	4.996	17.604	32	4.996	2m15.564
		6	5.921	33.852	64	5.921	2m25.816
		7	6.614	57.36	98	6.615	2m36.325
	500 - 2	2	1.999	3.128	4	1.999	6m0.768
		3	2.999	7.340	8	2.999	8m39.441
		4	3.999	10.816	16	3.999	10m33.013
		5	4.999	22.828	32	4.999	12m52.701
		6	5.997	39.844	64	5.997	15m20.654
		7	6.994	1m9.876	128	6.994	15m34.624
		8	7.966	2m6.796	256	7.985	17m43.237
		9	8.760	3m32.292	497	8.955	18m4.668
	10^6 - 2	2	2.	3.652	-	-	-
		3	3.	7.452	-	-	-
		4	4.	13.3	-	-	-
		5	4.999	25.24	-	-	-
		6	5.999	45.544	-	-	-
		7	6.999	1m26.22	-	-	-
		8	7.996	2m41.136	-	-	-
		9	8.939	4m31.396	-	-	-
		10	9.678	8m38.272	-	-	-
		11	10.06	15m8.224	-	-	-

Fig. 6. Results for MaxHNumeric and MaxHMarco.

information about the secret, since it indicates the similarity between `high` and `low`. This indicates a real-world vulnerability, where string `low` can be provided by a user via, e.g., a web form, and string `high` is the secret session information. They are concatenated and sent to the server. By observing compressed network packet size, a malicious user can reveal secret web session tokens [28]. This is another instance of a *segmented oracle* for which we computed the optimal attack.

LawDB is a network service application that provides access to records about law enforcement personnel. The application consists of 41 classes with 2844 line of codes, and uses the Netty library[2]. On the server side, LawDB stores all employee records in a database, and each employee is referenced with a unique ID. All IDs are loaded into a tree data structure when the server starts. There is a group of employees who works on clandestine activities; their IDs are restricted information. On the client side, there are several available operations, including a search for all IDs within a chosen range. Upon receiving the search request, the server delegates it to the tree data structure, which returns all the IDs in the range. If the ID is non-restricted, it is sent back to the client immediately in a UDP package; otherwise the server writes to an error log file, and does not send the restricted ID to the user. To analyze this example we populated the database with two concrete unrestricted IDs and one symbolic restricted ID, i.e. the secret h. The adversary performs the search operation by providing a symbolic range $[l_{min}, l_{max}]$.

Our techniques found a timing channel that is due to the fact that the response time of the server is noticeably longer when there are restricted IDs in the search range (due to the writing to the log file). Exploiting this timing channel, an adversary can perform an adaptive attack to discover a restricted ID. At a high level, this example is similar to our running example as the optimal attack involves narrowing down a *range* of secret values using repeated comparisons with low values. Specifically, the adversary makes a range request [min, max]. If the secret is in the range, then the execution time is longer. If the secret is outside the range then the time is slower. The adversary keeps making range queries smaller and smaller until it gets to size 1. Our techniques found this

[2]Netty library: http://netty.io/

attack automatically. We note that for this example we used an abstraction for the costs. First we preformed a symbolic analysis on one run of the program and we obtained 30 path conditions and 29 observables. We solved the path conditions, we obtained concrete test inputs and we executed the program (multiple times) on these inputs. Realtime measurements showed that only two group of observables are noticeably different. Therefore, we used abstraction to divide the costs into two intervals obtaining a binary search attack, which we validated by demonstrating it in operation.

ModPow implements modular exponentiation $b^e \texttt{ mod } m$; here base b is a public message, exponent e is a private key and modulus m is part of the public key. It is well-known that several implementations of modulo exponentiation have timing channels [29], [1]. We analyzed the ModPow example from [20] to compare with their work on non-adaptive attacks. Our techniques found a timing channel related to a reduction step present in the implementation. We note that this is a challenging example for symbolic execution as it involves complex, non-linear operations.

4) Results for MaxCC: The full approach, when it can finish, returns optimal attacks. However, it is expensive, since each analyzed clause encodes what amounts to k copies of path conditions obtained from a single program run. The greedy approach scales better but may not be optimal. See, e.g., results for running example from Fig. 1. At each attack step, the adversary provides an input, and can determine from the observation whether the secret is greater or smaller than her input. An optimal attack is a binary search in the secret's domain, which requires $\log_2(\texttt{DOMAIN})$ number of steps in the worst case. Fig. 4 confirms that, when $\texttt{DOMAIN} = 10$, the full approach reveals the whole secret in 4 steps ($\log_2(10) = 3.3$, note also that $maxObs = \texttt{DOMAIN}$ so full secret is revealed). In general, a k-step attack would reveal 2^k observables or the whole secret if its domain is less than 2^k. The attacks synthesized by the greedy approach are not optimal. For example, when $\texttt{DOMAIN} = 200$ the optimal strategy requires 8 steps to discover the whole secret; the greedy strategy requires 16 steps. However, it can synthesize this 16-step attack in less than 1 minute, while the full approach times out.

Note also that with the same number of steps, the full approach times out in small domains (200 - 500), but returns quickly when the domain is large (10^6). The reason is that when the domain is small some of the clauses are unsatisfiable, and UNSAT instances are usually expensive.

For `Arrays.equals` and `CRIME` the attacks generated by the greedy approach are as good as the attacks generated by the full approach. In particular the greedy approach synthesizes optimal attacks up to 5 steps (i.e. maxObs and leakage are the same for full and greedy at same number of steps). We conjecture that MaxCC greedy will always find the optimal attack for such examples that admit segmented oracles. The reason is that MaxSMT synthesizes a low input that can generate the maximum number of observables, and hence path conditions. This is only possible when the new low input is different from all the previous ones, therefore the whole procedure is a linear search on a segment, which is also the optimal attack in segmented oracles.

On the other hand, for `LawDB` and the illustrative example, MaxCC greedy does not generate the optimal attack, but still scales well and generates tight bounds on the leakage within a small number of steps as compared to the optimal attacks (see discussion in the next section).

For `ModPow`, our approach is much faster than the approach for non-adaptive attacks from [20]. For example, with $\texttt{MODULO} = 834443$ and $\texttt{DOMAIN} = 2^6$, our greedy approach is able to synthesize an attack that reveals the whole secret key in 4 steps. With the same configuration, the greedy approach in [20] times out even for a 2-step analysis.

5) Results for MaxHMarco and MaxHNumeric – greedy: Computing entropy is more expensive than computing channel capacity. In addition, MaxCC quickly generates the optimal attack for CRIME and `Array.equals` and so the more computationally expensive MaxH approaches are not necessary. On the other hand, we see that MaxCC does not generate the optimal attack for LawDB and the running example. Thus we apply the MaxH methods to these two examples using a 1-greedy configuration. Results are shown in Fig. 6.

In the illustrative example, MaxHMarco can synthesize the optimal strategy for DOMAIN up to 500, where MaxCC timed out, and MaxCC greedy is not able to synthesize the optimal attack. Furthermore MaxHMarco generates an attack for LawDB for a small domain where 7 steps are enough to reveal all 98 secret values. MaxCC is not able to analyze more than 4 steps, and MaxCC greedy needs 17 steps to reveal the full secret.

MaxHMarco relies on enumeration of partitions, so when there is a different partition for each public input it does not scale to large domains, and times out for a domain size of 10^6. On the other hand MaxHNumeric scales well and discovers attacks which leak only slightly less information for our examples. We also performed experiments for LawDB without abstraction, and we found that, as expected, the performance of both MaxHMarco and MaxHNumeric starts to degrade when the number of constraints increases. Thus the role of the abstraction is essential for the analysis of large systems and we plan to investigate it further in the future.

V. Related Work

There is large body of work on side-channel analysis, e.g. [29], [30], [1], [15], [31], [32], [33], [34], [20], [25], however few of them consider quantifying information leakage over multiple runs.

Köpf and Basin [15] were the first to show an information-theoretic model of adaptive multi-run side-channel attacks and to develop an automated technique to measure the remaining entropy after an attack. The technique is based on an enumeration algorithm (doubly-exponential in the number of attack steps); a greedy heuristic is also presented, to compute the remaining entropy of the secret after k steps. Similar to that work we use a partition-refinement approach to express the information that an attacker can gain about the secret. However the goal of our work is different: we focus on synthesizing the optimal attack while they aim to give theoretical bounds on the information leakage. Furthermore our approach is *symbolic*: while the previous work represents the knowledge about the secret in terms of sets of concrete values and computes adaptive attacks by enumerating over those values, we represent sets of secrets using mathematical constraints which encode succinctly much larger state spaces. This leads to important methodological differences allowing us to apply powerful optimization techniques over the constraints, without a need for explicit enumeration. A generalization of [15] to attack scenarios where secrets change over time is presented in [34]. That work uses probabilistic programming to model probabilistic, interactive systems and it is quite different from ours.

In previous work [20] we used symbolic execution and MaxSMT solving to synthesize multi-run attacks. However that work only considers *non-adaptive* attacks. In this paper we study adaptive attacks, i.e. a more powerful adversary, who selects the inputs based on previous observations, and the result of our synthesis is a strategy, i.e. different sequence of low inputs depending on the observables. Our results indicate that we can compute attacks much faster than in [20]; furthermore adaptive attacks are typically shorter. An important difference is that the approach from [20] only computes inputs that maximize channel capacity while in this work we also consider the harder problem of computing the inputs that maximize entropy (and other information theory measures). Another contribution is the use of an *abstraction* layer in the cost model.

In other recent work [25], we aimed to quantify adaptive attacks on programs with *segmented oracles*. In that work, we assumed the best attack was known and was encoded *manually* for analysis. In contrast we show here how to synthesize optimal attacks *automatically*. We have included the experiments on the largest case study (CRIME) from [25], and we show how we can compute automatically the optimal attack using a much larger input domain.

In the context of single-run analysis, the closet to our work is the Disco technical approach [4]. Disco is a framework for quantifying information leakage as defined by channel capacity and Shannon entropy, using a combination of model checking, quantifier elimination, and model counting. In theory, Disco can be generalized to multi-run analysis by applying it to the symbolic attack model that we defined in our paper. However, it remains to be seen how that would perform in practice, since, unlike our approach, Disco makes heavy use of self composition [35], which can be very expensive to check.

Furthermore, Disco computes what information can an attacker in principle gain (relation R [4]) but not the actual attack step represented by the low value. That is our main contribution: we synthesize a low value at each attack step that maximizes leakage wrt channel capacity and Shannon entropy. That low value can be used to build an attack.

Our MaxSMT method generates actual low inputs that maximize leakage even in the presence of non-linear constraints (see $modPow$ example). Disco does not compute low inputs and performs quantifier elimination applicable only to linear constraints. We do present two other methods that maximize leakage wrt Shannon entropy that work only for linear constraints; in our paper we implemented and compared three different ways of choosing lows adaptively. Besides the fact that they are not addressing the same problem, there are many methodological differences between Disco and our methods.

Another related work [11] proposes a technique for generating inputs that make the program violate a non-interference policy in the context of single-run attacks. The analysis considers only single-run attacks and is qualitative, not quantitative as our work, and thus can not be used or generalized to computing the adaptive attacks that maximize leakage.

Further related research includes the large body of work on the analysis of cache side channels [32], [33], focusing on a single round of observation. Our tool is built on top of SPF, a custom JVM with its own memory model, and can thus analyze some memory side-channels. However, currently our memory model does not have architecture-specific caches as in [32], [33].

VI. Conclusion and Future Work

We presented and evaluated symbolic analysis techniques for detecting vulnerabilities that are due to adaptive side-channel attacks and for synthesizing inputs that exploit the vulnerabilities. Our experiments show that the most scalable approach is MaxCC greedy. The approach also finds the optimal attack for side-channels with segmented oracles (`Arrays.equals` and CRIME). MaxH greedy methods are more expensive but they found optimal attacks for more involved, “binary” oracles. We are not aware of previous work that can automatically synthesize optimal attacks for complex examples such as LawDB. We also showed that our approach can scale better than previous work from [20]. Furthermore, we remark that the domains used in our experiments, e.g. 10^6 and 256^3, are beyond the capability of any concrete value enumeration-based methods, even for two runs.

Future work includes implementing a distributed version of our techniques, by creating and analyzing new parallel jobs with each observation made. We are also investigating approaches for model counting over non-linear constraints [36] to further extend the applicability of our techniques. We also plan to investigate the effects of non-determinism (including garbage collection) on the analysis. A related area is analysis and synthesis for side channels in the presence of noisy observations. Furthermore we plan to work on automated synthesis of prevention mechanisms to defend against attacks.

Acknowledgment

We thank Nikolaj Bjorner for useful discussions on MaxSMT and its generalization to the Maximal Satisfiable Subsets problem. We also thank the anonymous reviewers for their comments. This material is based on research sponsored by DARPA under agreement number FA8750-15-2-0087. The

U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

[1] D. Brumley and D. Boneh, "Remote Timing Attacks Are Practical," in *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, (Berkeley, CA, USA), pp. 1–1, USENIX Association, 2003.

[2] J. Kelsey, "Compression and Information Leakage of Plaintext," in *Revised Papers from the 9th International Workshop on Fast Software Encryption*, FSE '02, (London, UK, UK), pp. 263–276, Springer-Verlag, 2002.

[3] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, pp. 385–394, July 1976.

[4] M. Backes, B. Kopf, and A. Rybalchenko, "Automatic Discovery and Quantification of Information Leaks," in *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, SP '09, (Washington, DC, USA), pp. 141–153, IEEE Computer Society, 2009.

[5] J. Heusser and P. Malacaria, "Quantifying information leaks in software," in *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, (New York, NY, USA), pp. 261–269, ACM, 2010.

[6] Q.-S. Phan, P. Malacaria, O. Tkachuk, and C. S. Păsăreanu, "Symbolic Quantitative Information Flow," *SIGSOFT Softw. Eng. Notes*, vol. 37, pp. 1–5, Nov. 2012.

[7] V. Klebanov, N. Manthey, and C. Muise, "SAT-Based Analysis and Quantification of Information Flow in Programs," in *Quantitative Evaluation of Systems*, vol. 8054 of *Lecture Notes in Computer Science*, pp. 177–192, Springer Berlin Heidelberg, 2013.

[8] Q.-S. Phan and P. Malacaria, "Abstract Model Counting: A Novel Approach for Quantification of Information Leaks," in *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '14, (New York, NY, USA), pp. 283–292, ACM, 2014.

[9] Q.-S. Phan, P. Malacaria, C. S. Păsăreanu, and M. d'Amorim, "Quantifying Information Leaks Using Reliability Analysis," in *Proceedings of the 2014 International SPIN Symposium on Model Checking of Software*, SPIN 2014, (New York, NY, USA), pp. 105–108, ACM, 2014.

[10] Q.-S. Phan and P. Malacaria, "All-Solution Satisfiability Modulo Theories: applications, algorithms and benchmarks," in *Proceedings of the 2015 Tenth International Conference on Availability, Reliability and Security*, ARES '15, (Washington, DC, USA), IEEE Computer Society, 2015.

[11] Q. H. Do, R. Bubel, and R. Hähnle, "Exploit Generation for Information Flow Leaks in Object-Oriented Programs," in *ICT Systems Security and Privacy Protection: 30th IFIP TC 11 International Conference, SEC 2015, Hamburg, Germany, May 26-28, 2015, Proceedings*, (Cham), pp. 401–415, Springer International Publishing, 2015.

[12] R. Nieuwenhuis and A. Oliveras, "On SAT Modulo Theories and Optimization Problems," in *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing*, SAT'06, (Berlin, Heidelberg), pp. 156–169, Springer-Verlag, 2006.

[13] M. H. Liffiton and A. Malik, *Enumerating Infeasibility: Finding Multiple MUSes Quickly*, pp. 160–175. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013.

[14] C. S. Păsăreanu, W. Visser, D. Bushnell, J. Geldenhuys, P. Mehlitz, and N. Rungta, "Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis," *Automated Software Engineering*, pp. 1–35, 2013.

[15] B. Köpf and D. Basin, "An Information-theoretic Model for Adaptive Side-channel Attacks," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, (New York, NY, USA), pp. 286–296, ACM, 2007.

[16] P. Malacaria and H. Chen, "Lagrange multipliers and maximum information leakage in different observational models," in *Proceedings of the third ACM SIGPLAN workshop on Programming languages and analysis for security*, PLAS '08, (New York, NY, USA), pp. 135–146, ACM, 2008.

[17] G. Smith, "On the Foundations of Quantitative Information Flow," in *Proceedings of the 12th International Conference on Foundations of Software Science and Computational Structures*, FOSSACS '09, (Berlin, Heidelberg), pp. 288–302, Springer-Verlag, 2009.

[18] A. Filieri, C. S. Păsăreanu, and W. Visser, "Reliability analysis in Symbolic Pathfinder," in *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, (Piscataway, NJ, USA), pp. 622–631, IEEE Press, 2013.

[19] "Barvinok library." http://garage.kotnet.org/~skimo/barvinok/.

[20] C. S. Păsăreanu, Q.-S. Phan, and P. Malacaria, "Multi-run side-channel analysis using Symbolic Execution and Max-SMT," in *Proceedings of the 2016 IEEE 29th Computer Security Foundations Symposium*, CSF '16, (Washington, DC, USA), IEEE Computer Society, 2016.

[21] P. Malacaria, "Algebraic foundations for quantitative information flow," *Mathematical Structures in Computer Science*, vol. 25, pp. 404–428, 2 2015.

[22] W. Research, "Mathematica 11.0," 2016.

[23] S. Das and P. N. Suganthan, "Differential evolution: A survey of the state-of-the-art," *IEEE Trans. Evolutionary Computation*, vol. 15, no. 1, pp. 4–31, 2011.

[24] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on Information Theory*, vol. 23, pp. 337–343, May 1977.

[25] L. Bang, A. Aydin, Q.-S. Phan, C. S. Păsăreanu, and T. Bultan, "String Analysis for Side Channels with Segmented Oracles," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, (New York, NY, USA), pp. 193–204, ACM, 2016.

[26] "Google Keyczar library." https://github.com/google/keyczar.

[27] N. Lawson, "Timing attack in Google Keyczar library." https://rdist.root.org/2009/05/28/timing-attack-in-google-keyczar-library/, 2009.

[28] J. Rizzo and T. Duong, "The CRIME attack," Ekoparty Security Conference, 2012.

[29] P. C. Kocher, "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems," in *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '96, (London, UK, UK), pp. 104–113, Springer-Verlag, 1996.

[30] D. Agrawal, J. R. Rao, and P. Rohatgi, "Multi-channel attacks," in *Cryptographic Hardware and Embedded Systems - CHES 2003, 5th International Workshop, Cologne, Germany, September 8-10, 2003, Proceedings* (C. D. Walter, Ç. K. Koç, and C. Paar, eds.), vol. 2779 of *Lecture Notes in Computer Science*, pp. 2–16, Springer, 2003.

[31] S. Chen, R. Wang, X. Wang, and K. Zhang, "Side-Channel Leaks in Web Applications: A Reality Today, a Challenge Tomorrow," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, (Washington, DC, USA), pp. 191–206, IEEE Computer Society, 2010.

[32] B. Köpf, L. Mauborgne, and M. Ochoa, "Automatic quantification of cache side-channels," in *Proceedings of the 24th international conference on Computer Aided Verification*, CAV'12, (Berlin, Heidelberg), pp. 564–580, Springer-Verlag, 2012.

[33] G. Doychev, D. Feld, B. Köpf, L. Mauborgne, and J. Reineke, "CacheAudit: A Tool for the Static Analysis of Cache Side Channels," in *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, (Berkeley, CA, USA), pp. 431–446, USENIX Association, 2013.

[34] P. Mardziel, M. S. Alvim, M. W. Hicks, and M. R. Clarkson, "Quantifying Information Flow for Dynamic Secrets," in *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pp. 540–555, 2014.

[35] G. Barthe, P. R. D'Argenio, and T. Rezk, "Secure Information Flow by Self-Composition," in *Proceedings of the 17th IEEE workshop on Computer Security Foundations*, CSFW '04, (Washington, DC, USA), IEEE Computer Society, 2004.

[36] M. Borges, Q.-S. Phan, A. Filieri, and C. S. Păsăreanu, "Model-counting Approaches For Nonlinear Numerical Constraints," in *NASA Formal Methods: 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings*, pp. 131–138, Springer International Publishing, 2017.

Appendix

The implementation of the the Lempel-Ziv algorithm (LZ77) [24], used in the CRIME case study, is in Figure 7. As we have described, the string to be compressed is the concatenation of the public input and the secret. The adversary observes the length of the compressed string via, for example, a packet sniffer, and infers information about the secret.

```
public static byte[] compress(final byte[] in) throws IOException {

  StringBuffer mSearchBuffer = new StringBuffer(1024);
  String result = ””;

  String currentMatch = ””;
  int matchIndex = 0;
  int tempIndex = 0;
  int nextChar;
  for (int i = 0; i < in.length; i++){
    nextChar = in[i];

    tempIndex = mSearchBuffer.indexOf(currentMatch + (char)nextChar);
    if (tempIndex != -1) {
      currentMatch += (char)nextChar;
      matchIndex = tempIndex;
    }
    else {
      final String codedString = new StringBuilder().append(”˜”).append(matchIndex).append(”˜”)
          .append(currentMatch.length()).append(”˜”).append((char)nextChar).toString();
      final String concat = currentMatch + (char)nextChar;
      if (codedString.length() <= concat.length()) {
        result = result + codedString;
        mSearchBuffer.append(concat);
        currentMatch = ””;
        matchIndex = 0;
      }
      else {
        for (currentMatch = concat, matchIndex = -1; currentMatch.length() > 1 && matchIndex == -1;
                  currentMatch = currentMatch.substring(1, currentMatch.length()),
                                                matchIndex = mSearchBuffer.indexOf(currentMatch)) {
          result=result+currentMatch.charAt(0);
          mSearchBuffer.append(currentMatch.charAt(0));
        }
      }
      if (mSearchBuffer.length() <= 1024) {
        continue;
      }
      mSearchBuffer = mSearchBuffer.delete(0, mSearchBuffer.length() - 1024);
    }
  }
  if (matchIndex != -1) {
    final String codedString = new StringBuilder().append(”˜”).append(matchIndex)
                                      .append(”˜”).append(currentMatch.length()).toString();
    if (codedString.length() <= currentMatch.length()) {
      result = result + new StringBuilder().append(”˜”).append(matchIndex).append(”˜”)
                                      .append(currentMatch.length()).toString();
    }
    else {
      result = result + currentMatch;
    }
  }
  final byte[] bytes = result.getBytes();
  return bytes;
}
```

Fig. 7. CRIME