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]. 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 partitions 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 $S = (A, P, k, \text{cost}(\cdot))$, where the attacker $A$ generates values of $l$ for multiple executions of program $P$ in order to determine the secret $h$. Parameter $\text{cost}(\cdot)$ 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 $S = (A, P, k, \text{cost}(\cdot))$

1. seq ← ∅
2. for $i ← 1$ to $k$
3.   $l ← A(\text{seq})$
4.   $o ← \text{cost}(P(h, l))$
5. seq ← 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 $l ← A(i)$). Thus adaptive attacks can be more powerful, since the attacker can pick different low values at the 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


```c
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 \( \text{seq} \), \( \text{cost}(P(H, A(\text{seq}))) = \text{cost}(P(H', A(\text{seq}))) \).

**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 \( S = (A, P, k, \text{cost}()) \) 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 \cite{r16, r17} 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:

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

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 \( o^k_i = (o_1, o_2, ... o_k) \), let \( p(o^k_i) \) denote the probability of observing \( o^k_i \). Then the Shannon entropy is:

\[
H^k(P) = - \sum_{o^k_i} p(o^k_i) \log_2(p(o^k_i)) \tag{1}
\]

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 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 bound is put on the analysis depth. 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 \( \pi \) is a composition of paths \( \pi_1 \ldots \pi_k \), where each \( \pi_i \) is a path in the \( i \)-th invocation of \( P \). The cost of \( \pi \) is a k-observable, i.e., a sequence of k side-channel measurements made during the attack. Each path has a corresponding path condition, \( PC^k(h, l) \), 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 \( l \) 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 \( V \) 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 \( \bar{N}(s) \)

\[
A(seq) = V(\bar{N}(seq))
\]

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

\[
C_i(h, \bar{L}) = \bigvee_{cost(PC^k) = o^k_i} PC^k(h, \bar{L})
\]

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

\[
CC^k(P) = \log_2(|C|)
\]

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 \( \bar{o}^k_i \) is:

\[
p(\bar{o}^k_i) = \frac{\#(C_i(h, \bar{L}))}{\#D}
\]

Here \( \#(c) \) 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 \( \#D \) 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:

\[
H^k(P) = -\sum_{i=1,m} \frac{\#(C_i(h, \bar{L}))}{\#D} \log_2\left(\frac{\#(C_i(h, \bar{L}))}{\#D}\right)
\]

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 (1) and (2), with path conditions $h \leq l$ 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 (1) and a new symbolic value $l_2$ if the observed cost is (2). 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 (1, 2), one can determine that the secret satisfies the constraint $h \geq 1 \land h < l_1$.

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 (1) and 3 after (2). This corresponds to variable assignment: $l = 4, l_1 = 5, l_2 = 3$. After (1, 2) the attacker has learned that $h \geq 4 \land h < 5$ 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 (1, 2) the attacker has learned that $h \geq 4 \land h < 6$ which narrows down the possible values of $h$ to two (4 and 5) etc.

![Fig. 2. Symbolic tree for running example.](image)

**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 $\mathcal{M} \subseteq \mathcal{C}$ 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 $C_i(h, l)$ 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 $C_i(h, l)$ into $C_i(h_i, l_i)$ 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($P, k, cost(\cdot)$)**

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

**Procedure: ComputeConstraints($P, k, cost(\cdot)$)**

```plaintext
1 $\mathcal{O} \leftarrow \emptyset$, $\mathcal{C} \leftarrow \emptyset$
2 $\mathcal{PC} \leftarrow \text{SymEx}(S_{sym}(P, k, cost(\cdot)))$
3 foreach $\mathcal{PC}^k_i(h, l) \in \mathcal{PC}$ do
4 $\mathcal{O} \leftarrow \mathcal{O} \cup \{\text{cost}(\mathcal{PC}^k_i(h, l))\}$
5 foreach $\alpha_i^k \in \mathcal{O}$ do
6 $C_i(h, l) \leftarrow \bigcap_{\text{cost}(\mathcal{PC}^k_j(h, l)) = \alpha_i^k} \mathcal{PC}^k_j(h, l)$
7 $\mathcal{C} \leftarrow \mathcal{C} \cup \{C_i(h, l)\}$
8 return $\mathcal{C}$
```

**Procedure: MaxCC($\mathcal{C}$): Maximizing Channel Capacity**

```plaintext
1 $\mathcal{C} \leftarrow \text{Rename}(\mathcal{C})$
2 $(w, \bar{L}) \leftarrow \text{MaxSMT}(\mathcal{C})$
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 $l$ 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: \((1,1,1), (1,1,2), (1,2,1), (1,2,2), (2,1,1), (2,1,2), (2,2,1), (2,2,2)\). Each symbolic path yields different cost, thus each path condition corresponds to a clause, giving the following clauses (after renaming):

\[
\begin{align*}
{h_1} &\leq {l_1} \land {h_1} \geq {l_1} \land {h_1} \geq {l_1} \\
{h_2} &\leq {l_2} \land {h_2} \geq {l_2} \land {h_2} \leq {l_1} \\
{h_3} &\leq {l_3} \land {h_3} \leq {l_1} \land {h_3} \leq {l_2} \\
{h_4} &\leq {l_4} \land {h_4} \leq {l_1} \land {h_4} \leq {l_2} \\
{h_5} &\leq {l_5} \land {h_5} \leq {l_2} \land {h_5} \leq {l_3} \\
{h_6} &\leq {l_6} \land {h_6} \leq {l_2} \land {h_6} \leq {l_4} \\
{h_7} &\leq {l_7} \land {h_7} \leq {l_2} \land {h_7} \leq {l_3} \\
{h_8} &\leq {l_8} \land {h_8} \leq {l_2} \land {h_8} \leq {l_4}
\end{align*}
\]

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 observables after 3 steps, is 6, which is equal to the domain of satisfiable clauses, corresponding to maximum number of pruning necessary – in our implementation we exploit such situations that the partitions on the right sub-tree in the figure already attacker can always guess the secret for this example. Note the blocks in the partition have size 1, confirming that the define the partition on the secret induced by this attack. All 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.](image)

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, \(L = (V(l_1), V(l_2), \ldots)\). Our goal is to choose \(L\) which maximizes the Shannon entropy given that choice of \(L\), which we denote \(H^k(P|L)\). 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 \(L\) with objective function \(H^k(P|L)\). 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(\(P, k, \text{cost}(\cdot)\))

1. \(C \leftarrow \text{ComputeConstraints}(P, k, \text{cost}(\cdot))\)
2. \(L \leftarrow \text{MaxH}(C, \text{true})\)
3. return \((H^k(P|L), L)\)

Algorithm 2 AdaptiveMaxLeakH, outlines our approach. We first use symbolic execution to compute the set of clauses \(C\) which partitions the input. We then compute the value of \(L\) 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 \(C\) be a set of clauses of the form \(C(h, \bar{L})\) as before, where each clause has the weight 1. Set \(M \subseteq C\) is a maximal satisfiable subset (MSS) if (the conjunction of all clauses in) \(M\) is satisfiable and \(\forall C \in C \setminus M : M \cup \{C\}\) 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 \(\{C_1(h, \bar{L}), C_2(h, \bar{L}), \ldots\}\) 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 \(M\) 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($C, D$): Maximizing Entropy

1. $C \leftarrow \text{Rename}(C)$
2. $M \leftarrow \text{Marco}(C)$
3. $\text{leak}_\text{max} \leftarrow 0$
4. foreach MSS $\in M$
   
   5. $\varphi(l) \leftarrow \text{true}$
   
   6. foreach $C_l(h_i, \bar{l}) \in \text{MSS}$ do
      
      7. $\varphi(l) \leftarrow \varphi(l) \land C_l(h_i, \bar{l})$
      
      while isSAT($\varphi(l)$) do
      
      9. $L \leftarrow \text{getModel}(\varphi(l))$
      
      10. foreach $C_l(h_i, \bar{l}) \in \text{MSS}$ do
      
      11. $p(o^k_l) \leftarrow \mathbb{Z}(C_l(h_i, \bar{l}))/\mathbb{Z}D$
      
      12. $\text{leak} \leftarrow -\sum p(o^k_l) \log_2(p(o^k_l))$
      
      13. if $\text{leak}_\text{max} < \text{leak}$ then
      
      14. $\text{leak}_\text{max} \leftarrow \text{leak}$
      
      15. $L_{\text{max}} \leftarrow L$
      
      16. $\varphi(l) \leftarrow \varphi(l) \land \neg \varphi(L)$
      
return $L_{\text{max}}$

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

$$\varphi(l) \leftarrow \varphi(l) \land \neg \varphi(L)$$

We discover all the partitions of the secret associated with this MSS, when $\varphi$ 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 $M$.

Proof: By construction $M$ contains all MSSs. This means that any other sub-set $S \subseteq C$ such that $S \not\subset M$ is included by an MSS, i.e., $\exists M \in M$ 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

$$\text{leak} \leftarrow -\sum p(o^k_l) \log_2(p(o^k_l))$$

with

$$\text{leak} \leftarrow \sum b_l (b_l/2D) (b_l+1)/2$$

where $b_l = \mathbb{Z}(C_l(h_i, \bar{l}))$ and replace the test if $\text{leak}_\text{max} < \text{leak}$ with if $\text{leak}_\text{max} > \text{leak}$. 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 $\text{leak}_\text{max} < \text{leak}$ 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 $M$.

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 $C = h \geq l, h < l$. Running the Marco algorithm yields only one maximal satisfiable subset, $C$ itself. Then $\varphi(l)$ is $h_1 \geq l \land h_2 < l$. Suppose solving $\varphi$ 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: $\varphi(l) :: h_1 \geq l \land h_2 < l \land \neg (h_1 \geq 2 \land h_2 < 2)$. 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 $C_l(h, \bar{l})$ is a function $F_l(\bar{l})$ that computes the number of possible secrets $h$ that satisfy $C_l$, 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 $C_1$ or cost $C_2$ is observed. Then there are 4 possible constraints over the vector $l = (l, l_1, l_2)$ corresponding to the leaves of the symbolic attack tree.

$$C_1 = h < \ell \land h < \ell_1 \quad C_2 = h < \ell \land h \geq \ell_1 \quad C_3 = h \geq \ell \land h < \ell_2 \quad C_4 = h \geq \ell \land h \geq \ell_2$$

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 $1 \leq l \leq 6 \land l \leq l_1$ then there are exactly $l - 1$ values of $h$ that satisfy $C_1$. Symmetrically, if $1 \leq l_1 \leq 6 \land l < l_1$ 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.)

\[
F_i(\bar{\bar{l}}) = \begin{cases} 6 & : l > 6 \land l_1 > 6 \\ l - 1 & : 1 \leq l \leq 6 \land l \leq l_1 \\ l_1 - 1 & : 1 \leq l_1 \leq 6 \land l_1 < l \\ \end{cases}
\]

\[
F_2(l) = \begin{cases} 6 & : l_1 < 1 \land 6 < l \\ l - l_1 & : 1 \leq l \leq l_1 \\ l - 1 & : 1 < l_1 \leq l \\ 7 - l & : 1 \leq l \leq 6 < l \\ \end{cases}
\]

\[
F_3(l) = \begin{cases} 6 & : l < 1 \land 6 < l_2 \\ l_2 - l & : 1 \leq l \leq l_2 \\ l_2 - 1 & : 1 < l \leq l_2 \\ 7 - l & : 1 \leq l \leq 6 < l_2 \\ \end{cases}
\]

\[
F_4(l) = \begin{cases} 6 & : l < 1 \land l_2 < 1 \\ 7 - l & : 1 \leq l \leq 6 \land l_2 < l \\ 7 - l_2 & : 1 \leq l_2 \leq 6 \land 1 \leq l_2 \\ \end{cases}
\]

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

Using each \( F_i(l) \) we compute the probability of an observation sequence given the values of the low inputs as \( p(o^k_i|l) = F_i(l) / \#D \). We then plug these symbolic probability functions into Equation [1]

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

Then, the attack synthesis can be stated as a non-linear objective function maximization problem, defined by \( \bar{L} = \arg \max\mathcal{H}^k(P|\bar{l}) \). 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(C, D): Maximizing Entropy**

1. foreach \( C_i \in C \) do
2. \( F_i(\bar{\bar{l}}) \leftarrow Barvinok(C_i, \bar{\bar{l}}, h) \)
3. \( p(o^k_i|\bar{l}) \leftarrow F_i(\bar{\bar{l}}) / \#D \)
4. \( \mathcal{H}^k(P|\bar{l}) \leftarrow - \sum_{i=1}^{m} p(o^k_i|\bar{l}) \log_2(p(o^k_i|\bar{l})) \)
5. \( \bar{L} = N\text{Maximize}(\mathcal{H}^k(P|\bar{l})) \)
6. return \( \bar{L} \)

For the example from Figure 1, maximizing \( \mathcal{H}(P|\bar{l}) \) results in the assignment \( \bar{L} = \{4, 2, 5\} \) for symbolic inputs \( \{l, l_1, l_2\} \). 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(\( P, k, \text{cost}() \)) 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 [22] 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
### Illustrative Example in Fig. 1

<table>
<thead>
<tr>
<th>Domain</th>
<th>Steps</th>
<th>Full maxObs</th>
<th>Full CC</th>
<th>Full time</th>
<th>I-greedy maxObs</th>
<th>I-greedy CC</th>
<th>I-greedy time</th>
</tr>
</thead>
<tbody>
<tr>
<td>LawDB</td>
<td>100-2</td>
<td>2</td>
<td>4</td>
<td>2.750</td>
<td>4</td>
<td>4</td>
<td>2.441</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>8</td>
<td>3</td>
<td>19.493</td>
<td>7</td>
<td>2.007</td>
<td>3.244</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>16</td>
<td>3</td>
<td>3m8.783</td>
<td>11</td>
<td>3.459</td>
<td>4.160</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>17</td>
<td>4.087</td>
<td>5.342</td>
</tr>
<tr>
<td></td>
<td>17</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>98</td>
<td>6.615</td>
<td>21.010</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>4</td>
<td>2</td>
<td>2.762</td>
<td>4</td>
<td>2</td>
<td>2.579</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>8</td>
<td>3</td>
<td>19.493</td>
<td>8</td>
<td>3</td>
<td>3.866</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>16</td>
<td>3M21.688</td>
<td>16</td>
<td>4</td>
<td>5.844</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>32</td>
<td>5</td>
<td>9.212</td>
</tr>
<tr>
<td></td>
<td>17</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>6070</td>
<td>12.567</td>
<td>18m31.246</td>
</tr>
</tbody>
</table>

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

we implemented SPF listeners to monitor the 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 `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 $o_{\text{min}}$ and $o_{\text{max}}$ 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 to $(n-1)^{th}$, where $n$ is a user supplied parameter. Each interval has equal size, $d = \frac{o_{\text{max}} - o_{\text{min}}}{n}$. We then map all the costs $o_{\text{obs}}$ such that $o_{\text{min}} + i \times d \leq o_{\text{obs}} < o_{\text{min}} + (i+1) \times d$ to the same interval $i$ ($o_{\text{max}}$ 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 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 provided to us by DARPA [25], 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.

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.Arrays.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 [26]. Our techniques computed the optimal attack automatically. We note that the use of `java.util.Arrays.equals` in Google’s Keyczar library [27] led to a vulnerability that could allow an adversary to forge signatures for data that was “signed” with the default SHA-1 HMAC algorithm [28].

CRIME is inspired by the “Compression Ratio Infoleak Made Easy” attack [29]. We analyzed procedure: `compress(high.concat(low))`. Our techniques found a space side channel: the size of the compressed string leaks information about the secret, since it indicates the similarity between `high` and `low`. This indicates a real-world vulnera-

<table>
<thead>
<tr>
<th>Case Study</th>
<th>MODULO</th>
<th>DOMAIN</th>
<th>Steps</th>
<th>Full maxObs</th>
<th>Full CC</th>
<th>Full time</th>
<th>1-greedy maxObs</th>
<th>1-greedy CC</th>
<th>1-greedy time</th>
</tr>
</thead>
<tbody>
<tr>
<td>ModPow [20]</td>
<td>1717</td>
<td>$2^3 - 1$</td>
<td>2</td>
<td>7</td>
<td>2.807</td>
<td>7.679</td>
<td>7</td>
<td>2.807</td>
<td>3.572</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$2^3 - 1$</td>
<td>2</td>
<td>15</td>
<td>3.906</td>
<td>1m46.236</td>
<td>14</td>
<td>3.807</td>
<td>14.199</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$2^5 - 1$</td>
<td>2</td>
<td>3</td>
<td>-</td>
<td>-</td>
<td>25</td>
<td>4.644</td>
<td>1m6.624</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$2^5 - 1$</td>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>30</td>
<td>4.906</td>
<td>5m36.462</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$2^5 - 1$</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>31</td>
<td>4.954</td>
<td>8m5.695</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$2^9 - 1$</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>34</td>
<td>5.087</td>
<td>9m47.837</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$2^9 - 1$</td>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>53</td>
<td>5.728</td>
<td>37m31.318</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$2^9 - 1$</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ModPow [20]</td>
<td>834443</td>
<td>$2^3 - 1$</td>
<td>2</td>
<td>7</td>
<td>2.807</td>
<td>10.704</td>
<td>7</td>
<td>2.807</td>
<td>3.624</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$2^3 - 1$</td>
<td>2</td>
<td>15</td>
<td>3.906</td>
<td>8m67.264</td>
<td>14</td>
<td>3.807</td>
<td>14.292</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$2^5 - 1$</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>25</td>
<td>4.644</td>
<td>1m26.862</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$2^5 - 1$</td>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>30</td>
<td>4.906</td>
<td>3m7.683</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$2^5 - 1$</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>31</td>
<td>4.954</td>
<td>3m49.175</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$2^9 - 1$</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>46</td>
<td>5.524</td>
<td>4m24.405</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$2^9 - 1$</td>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>62</td>
<td>5.954</td>
<td>25m9.132</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$2^9 - 1$</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>63</td>
<td>5.977</td>
<td>35m53.526</td>
</tr>
<tr>
<td>ModPow [20]</td>
<td>1964903306</td>
<td>$2^3 - 1$</td>
<td>2</td>
<td>7</td>
<td>2.807</td>
<td>13.124</td>
<td>7</td>
<td>2.807</td>
<td>3.269</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$2^3 - 1$</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>14</td>
<td>3.807</td>
<td>17.063</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$2^5 - 1$</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>25</td>
<td>4.644</td>
<td>1m24.405</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$2^5 - 1$</td>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>30</td>
<td>4.906</td>
<td>4m1.789</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$2^5 - 1$</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>31</td>
<td>4.954</td>
<td>4m27.231</td>
</tr>
</tbody>
</table>

Fig. 5. Results for MaxCC on ModPow [20].
<table>
<thead>
<tr>
<th>Case Study</th>
<th>DOMAIN</th>
<th>Steps</th>
<th>MaxHNumeric</th>
<th>MaxHMarco</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Leakage (bits)</td>
<td>time</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>8</td>
<td>7.207</td>
<td>44.876</td>
</tr>
<tr>
<td></td>
<td>300</td>
<td>9</td>
<td>7.560</td>
<td>69.383</td>
</tr>
<tr>
<td></td>
<td>400</td>
<td>9</td>
<td>7.743</td>
<td>1m55.212</td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>9</td>
<td>7.800</td>
<td>1m24.068</td>
</tr>
<tr>
<td>Illustrative Example in Fig. 1</td>
<td>10^6</td>
<td>10</td>
<td>8.172</td>
<td>3m15.000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11</td>
<td>8.303</td>
<td>4m55.088</td>
</tr>
<tr>
<td></td>
<td></td>
<td>12</td>
<td>8.357</td>
<td>7m12.280</td>
</tr>
<tr>
<td></td>
<td></td>
<td>13</td>
<td>8.371</td>
<td>9m34.512</td>
</tr>
<tr>
<td></td>
<td></td>
<td>14</td>
<td>8.376</td>
<td>12m20.844</td>
</tr>
<tr>
<td>LawDB</td>
<td>100 . 2</td>
<td>2</td>
<td>1.999</td>
<td>2.552</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>2.999</td>
<td>4.688</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>3.998</td>
<td>10.284</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>4.996</td>
<td>17.604</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
<td>5.921</td>
<td>33.852</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7</td>
<td>6.614</td>
<td>57.36</td>
</tr>
<tr>
<td></td>
<td>500 . 2</td>
<td>2</td>
<td>1.999</td>
<td>3.128</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>2.999</td>
<td>7.340</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>3.999</td>
<td>10.816</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>4.999</td>
<td>22.828</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
<td>5.997</td>
<td>39.844</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7</td>
<td>6.994</td>
<td>1m9.876</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>7.966</td>
<td>2m6.796</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9</td>
<td>8.760</td>
<td>3m32.292</td>
</tr>
<tr>
<td></td>
<td>10^9 . 2</td>
<td>2</td>
<td>2.</td>
<td>3.652</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>3.</td>
<td>7.452</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>4.</td>
<td>13.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>4.999</td>
<td>25.24</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
<td>5.999</td>
<td>45.544</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7</td>
<td>6.999</td>
<td>1m26.22</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>7.996</td>
<td>2m41.136</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9</td>
<td>8.939</td>
<td>4m31.396</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10</td>
<td>9.678</td>
<td>8m38.272</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11</td>
<td>10.06</td>
<td>15m8.224</td>
</tr>
</tbody>
</table>

Fig. 6. Results for MaxHNumeric and MaxHMarco.

bility, 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 [29]. 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 [30]. 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 $[l_{\text{min}}, l_{\text{max}}]$.

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 $[\text{min}, \text{max}]$. 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 attack automatically. We note that for this example we used an abstraction for the costs. First we performed 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 pro-
gram (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 $b^e \mod m$;
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 [31], [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 $\log_2(\text{DOM\textsc{ain}})$ number of steps in
the worst case. Fig. 4 confirms that, when $\text{DOM\textsc{ain}} = 10$, the full
approach reveals the whole secret in 4 steps ($\log_2(10) = 3.3$,
note also that $\text{maxObs} = \text{DOM\textsc{ain}}$ 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 $\text{DOM\textsc{ain}} = 200$ 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 $\text{MODULO}
= 834443$ and $\text{DOM\textsc{ain}} = 2^6$, 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 Arrays.equals and so the more com-
putationally 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 $\text{DOM\textsc{ain}}$ 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.
[31], [32], [11], [15], [33], [34], [35], [36], [20], [26], 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 [36]. 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 [26], 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 [26], and we show how we can compute automatically the optimal attack using a much larger input domain.

In the context of single-run analysis, a related work is the Disco 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 [37], 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 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 [54], [55], 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 [34], [35].

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 256^4, 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 [38] 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 copy-right notation thereon.
REFERENCES


APPENDIX
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 += 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 += currentMatch.charAt(0);
                    mSearchBuffer.append(currentMatch.charAt(0));
                }
            }
        }
    }
    if (mSearchBuffer.length() <= 1024) {
        continue;
    }
    mSearchBuffer = mSearchBuffer.delete(0, mSearchBuffer.length() - 1024);
}

for (matchIndex != -1) {
    final String codedString = new StringBuilder().append("˜").append(matchIndex).append("˜").append(currentMatch.length()).append("˜").append(currentMatch.length()).append("˜").append((char)nextChar).toString();
    if (codedString.length() <= currentMatch.length()) {
        result += codedString;
        mSearchBuffer.append(concat);
    } else {
        result += new StringBuilder().append("˜").append(matchIndex).append("˜").append(currentMatch.length()).append("˜").append(currentMatch.length()).append("˜").append((char)nextChar).toString();
    }
}

final byte[] bytes = result.getBytes();
return bytes;
}

Fig. 7. The implementation of the Lempel-Ziv compression algorithm (LZ77) [24] which is used in the CRIME case study. 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.