String Analysis for Side Channels with Segmented Oracles

Lucas Bang\textsuperscript{1}, Abdulbaki Aydin\textsuperscript{1}, Quoc-Sang Phan\textsuperscript{2}, Corina S. Păsăreanu\textsuperscript{2,3}, Tevfik Bultan\textsuperscript{1}
\textsuperscript{1} University of California, Santa Barbara, CA, USA
\textsuperscript{2} Carnegie Mellon University, Moffet Field, CA, USA
\textsuperscript{3} NASA Ames Research Center, Moffet Field, CA, USA
{bang,baki,bultan}@cs.ucsb.edu sang.phan@sv.cmu.edu corina.s.pasareanu@nasa.gov

ABSTRACT

We present an automated approach for detecting and quantifying side channels in Java programs, which uses symbolic execution, string analysis and model counting to compute information leakage for a single run of a program. We further extend this approach to compute information leakage for multiple runs for a type of side channels called segmented oracles, where the attacker is able to explore each segment of a secret (for example each character of a password) independently. We present an efficient technique for segmented oracles that computes information leakage for multiple runs using only the path constraints generated from a single run symbolic execution. Our implementation uses the symbolic execution tool Symbolic PathFinder (SPF), SMT solver Z3, and two model counting constraint solvers LattE and ABC. Although LattE has been used before for analyzing numeric constraints, in this paper, we present an approach for using LattE for analyzing string constraints. We also extend the string constraint solver ABC for analysis of both numeric and string constraints, and we integrate ABC in SPF, enabling quantitative symbolic string analysis.

CCS Concepts

\begin{itemize}
\item Software and its engineering \rightarrow Formal software verification;
\item Security and privacy \rightarrow Logic and verification;
\end{itemize}

Keywords

Side-channel analysis; Symbolic execution; String constraints

1. INTRODUCTION

Since computers are used in every aspect of modern life, many software systems have access to secret information such as financial and medical records of individuals, trade secrets of companies and military secrets of states. Confidentiality, a core computer security attribute, dictates that, a program that manipulates secret information should not reveal that information. This can be hard to achieve if an attacker is able to observe different aspects of program behavior such as execution time and memory usage.

Side-channel attacks recover secret information from programs by observing non-functional characteristics of program executions such as time consumed, number of memory accessed or packets transmitted over a network. In this paper, we propose an automatic technique for side-channel analysis. Our technique uses symbolic execution for the systematic analysis of program behaviors under different input values. Furthermore, we use model counting over the constraints collected with symbolic execution to quantify the leakage of the detected side channels.

We present specialized techniques for segmented oracle side channels in which an attacker is able to explore each segment of a secret, for example each character of a password, independently. Our technique can answer questions such as “what is the probability of discovering a password in k runs” or “what is the leakage (in the number of bits) after k runs” through side channels.

The widespread use of web-based applications have resulted in a greater need for analysis techniques targeted at string manipulating programs to ensure better security. However, classic testing approaches, such as guided black-box and random testing are not capable of reliably detecting malicious behaviors, simply because the domain of string inputs is too large. In contrast, symbolic execution can explore multiple inputs all at once, through the systematic collection and solving of symbolic constraints. A key challenge that we address in this work is to perform constraint solving and model counting efficiently over a combination of string and numeric constraints. Towards this end we investigate a set of complementary techniques for symbolic quantitative string analysis. We implemented these techniques in the Symbolic PathFinder (SPF) tool \cite{schuster2013symbolic}.

Our contributions can be summarized as follows: 1) Single-run side-channel analysis using SPF that computes the information leakage in terms of Shannon entropy using probabilistic symbolic execution and listeners that track the observable values such as execution time, file size, or memory usage. 2) Two types of multi-run side-channel analysis for segmented oracles based on a best-adversary model. The first approach composes the adversary model and the function under analysis within a loop and conducts the multi-run analysis on the composed system. However, this approach leads to path explosion. We also present a second, more efficient, approach for multi-run side-channel analysis for segmented oracles that uses path constraints generated for only
a single-run symbolic execution of the function. 3) We extend SPF to enable analysis of Java programs that manipulate strings using two approaches. One of them traces the implementations of string manipulation functions and treats strings as bounded arrays of characters that are represented as bit-vectors, and checks satisfiability of path constraints using the SMT solver Z3. The second approach generates constraints in the theory of strings and uses the string constraint solver ABC. 4) We use two model counting constraint solvers for computing information leakage. One of them is a model counter for numeric constraints called LattE, which has been used for analyzing numeric constraints in SPF before. In this paper we extend the SPF+LattE framework to the analysis of string constraints by viewing strings as arrays of characters. We also integrate the model-counting string constraint solver ABC with SPF. We further implement an extension to ABC that enables model counting for numeric constraints. With this extension to ABC can perform model counting for combinations of numeric and string constraints. 5) We conduct experiments on two side-channel examples, demonstrating the performance of different approaches.

Our approach applies to functions which behave as segmented oracles. For example, time-based segmented oracles can result from library functions that use early termination optimizations for string, array, and memory equality comparisons, which are present in many programming languages including C, C++, Java, Python, Ruby, PHP, and Node.js [?, ?, ?, ?]. As described in [?], these types of library functions have enabled real-world segmented oracle attacks against the Xbox 360 operating system [?], and the hash-message authentication code (HMAC) comparisons in the Google Keyczar cryptographic library [?] and the open authorization standards OAuth [?, ?] and OpenID [?]. We demonstrate the applicability of our approach given in Section 4.2 on a password verification function as an example of an early termination segmented oracle and this analysis applies equally well to the other examples just described. On the other hand, size-based segmented oracles can arise from text compression functions [?] resulting in leakage of confidential web-session information by measuring the sizes of files and network communications [37]. We demonstrate our approach for this type of segmented oracles using LZ77 compression [34]. Although the approach we present in Section 4.2 requires an ordered traversal of the secret’s segments (which is the case for all the examples listed above), we believe that, in the future, it can be generalized to handle oracles which do not require a specific ordering of segments.

2. SIDE-CHANNEL ANALYSIS

Consider a password-based authentication function. The password checking function has two inputs: 1) a password, which is secret, and 2) a user input, which is public. The function should compare the password and the user input and return true if the input matches the password and return false otherwise; it should not leak any information about the password if the input does not match.

Let us consider two password checking functions $F_1$ and $F_2$ whose implementations are shown in Figures 1 and 2 respectively. We assume that functions $F_1$ and $F_2$ are executed on inputs $h$ and $l$, where we follow the typical notation used in the security literature: $h$ denotes the high value, i.e., the secret password, and $l$ denotes the low value, i.e., the public input that the function compares with the password.

```
public int F1 (char[] h, char[] l)
  for (int i = 0; i < h.length; i++)
    if (h[i] != l[i]) return false;
  return true;
```

Figure 1: Password checking function $F_1$.

```
public int F2 (char[] h, char[] l)
  matched = true;
  for (int i = 0; i < h.length; i++)
    if (h[i] != l[i]) matched = false;
  return matched;
```

Figure 2: Password checking function $F_2$.

Both functions return true or false indicating if the input ($l$) matches the secret ($h$). Note however that the functions may leak some information about the secret through side channels, in this case an adversary may infer some information about the secret $h$ by measuring the execution time (as explained below). In general, let us assume that function $F(h,l)$ returns an observable value $o$ which represents the side-channel measurements of the adversary after executing $F(h,l)$. The observable value $o$ can be one of a set of observable values $O$. We assume that the observable values are noiseless, i.e., multiple executions of the program with the same input value will result in the same observable value.

For the functions $F_1$ and $F_2$ above, let us use the execution time as an observable. For function $F_1$ this will result in $n + 1$ observable values where $n$ is equal to the length of $h$, i.e., $o \in \{o_0, o_1, ... , o_n\}$, since function $F_1$ will have a different execution time based on the length of the common prefix of $h$ and $l$. If $h$ and $l$ have no common prefix, then $F_1$ will have the shortest execution time (let us call this observable value $o_0$) since the loop body will be executed only once (assuming the password length is not zero). If $h$ and $l$ have a common prefix of one character (and assuming that password length is greater than or equal to two), then $F_1$ will have a longer execution time since the loop body will be executed twice (let us call this observable value $o_1$). In fact, for each different length of the common prefix of $h$ and $l$, the execution time of $F_1$ will be different. Let observable value $o_i$ denote the execution time of $F_1$ for the common prefix of size $i$. Note that $o_n$ corresponds to the case where $h$ and $l$ completely match where $n$ is the length of the password. On the other hand, execution time of $F_2$ is always the same, so $F_2$ does not have a side channel.

Side-channel analysis can be used to answer following type of questions: Are $F_1$ and $F_2$ leaking information about the secret through side channels, and, if so, how much? Based on the above discussion, we can see that $F_1$ is leaking information about $h$ even when $h$ and $l$ do not completely match. By observing the execution time of $F_1$, an adversary can deduce the length of the common prefix of $h$ and $l$. $F_2$ on the other hand leaks no information through the side channel. Note that an attacker learns that $h$ is not equal to the value of $l$. However, the information leakage for $F_2$ is pretty small compared to the information leakage by $F_1$ (which has the timing side channel). For example, assuming the secret is a four digit PIN, an attacker needs at most $10^4$ tries to guess the password using $F_2$ but at most 40 tries using $F_1$ (as the adversary can try to first guess the first
Entropy Computation

Shannon entropy is a well-known information theoretic concept for measuring the expected amount of information contained in a message. The observables produced by a function $F$ can be considered the messages that an adversary receives by executing $F$; they correspond to messages about the secret. Hence, we can use the Shannon entropy to measure the amount of information conveyed by each execution of $F$, i.e., the amount of information leaked by function $F$. We define the Shannon entropy of a function $F$ as:

$$\mathcal{H}(F) = -\sum_{o_i \in O} p(o_i) \times \log_2(p(o_i))$$

where $p(o_i)$ is the probability of observing the value $o_i$ after executing $F$. In order to compute the Shannon entropy, we need to compute the probability of observing each value $o_i$. We can compute these probabilities using probabilistic symbolic execution with model counting [12].

Symbolic Execution (SE) [21] is a well-known program analysis technique that executes the program on symbolic, rather than concrete, inputs and computes the program effects as functions in terms of these symbolic inputs. Symbolic execution also computes and maintains a path condition, $PC$, which is a conjunction of constraints over the symbolic inputs that characterizes those inputs that follow each path through the program.

In our analysis we perform symbolic execution (where both $h$ and $l$ are symbolic) to systematically analyze all paths through the code. In this way, we collect each path condition and corresponding observable as an ordered pair $(o_i, PC_i)$. For example, symbolic execution of function $F_1$ results in a set of $n + 1$ path conditions, each with a different time observation: $\{(o_0, PC_0), (o_1, PC_1), \ldots (o_n, PC_n)\}$, with $o_0 < o_1 < \ldots < o_{n+1}$, and path conditions of the form:

- $PC_0 \equiv h[0] \neq l[0]$
- $PC_1 \equiv h[0] = l[0] \wedge h[1] \neq l[1]$
- $PC_2 \equiv h[0] = l[0] \wedge h[1] = l[1] \wedge h[2] \neq l[2]$
  
  $\vdots$
- $PC_{n-1} \equiv h[0] = l[0] \wedge h[1] = l[1] \wedge \ldots \wedge h[n-1] \neq l[n-1]$
- $PC_n \equiv h[0] = l[0] \wedge h[1] = l[1] \wedge \ldots \wedge h[n] = l[n]$

Each path condition $PC_i$ encodes the fact that a prefix of the public input $l$ matches a prefix of the secret $h$. For example $(o_1, PC_1)$ indicates that the first character in the public input matches the first character in the secret and the second character does not match. For the largest observable, we see that $(o_n, PC_n)$ indicates that the public and private inputs match on all segments.

We can compute the probability $p(o_i)$ for each observable value $o_i$, using model counting over the path conditions, in the following way. Let $D$ denote the input domain (i.e., the set of possible values for $h$ and $l$, assumed to be finite) and let $|D|$ denote the size of the input domain. We write $|PC|$ to denote the number of solutions over $D$ that satisfy the path constraint $PC$. We can compute $|PC|$ using a model counting constraint solver [27, 3]. Assuming a uniform distribution for $h$ and $l$ the probability of observing $o_i$ is $p(o_i) = |PC_i|/|D|$. The probability of each observable is computed as:

$$p(o_i) = \frac{|PC_i|}{|D|}$$

For function $F_2$, there are only two observable values through the main channel, i.e., the boolean values returned by the function, and the corresponding path constraints are:

- $\neg(h[0] = l[0] \wedge h[1] = l[1] \wedge \ldots h[n-1] = l[n-1])$
- $h[0] = l[0] \wedge h[1] = l[1] \wedge \ldots h[n-1] = l[n-1]$

Figure 3 shows the Shannon entropy computed for $F_1$ and $F_2$ as described above using probabilistic symbolic execution and model counting. Note that as the size of the password increases, the entropy gets very close to 0 for function $F_2$. So, for a reasonable sized password, $F_2$ does not leak information. However, for $F_1$ we observe that the information leaked remains around 1 bit even if we keep increasing the length of the password. Independent of the size of the password, $F_1$ leaks information about the first digit of the password due to the timing side channel.

The analysis we presented above computes the amount of information leaked by a single execution of a function. We can also easily determine the amount of initial information in the system by assuming that $h$ is picked using a uniform distribution from the domain $D_h$. Then the amount of information in the system initially is:

$$H(h) = -\sum_{v \in D_h} \frac{1}{|D_h|} \times \log_2(\frac{1}{|D_h|}) = \log_2(|D_h|)$$

An execution of the function leaks the amount of information given by the Shannon entropy of the function $H(F)$ and the remaining entropy in the system is $H(h) - H(F)$.

An interesting question to answer is the following: How many tries would it take an adversary to figure out the password? We can try to estimate the attack sequence length using the information leakage. When the amount of information in the system reaches zero, then, we can conclude that the adversary has figured out the password.

Based on the amount of initial information and the Shannon entropy for the function, we can try to estimate the amount of runs it would take an adversary to determine the secret. However, this analysis would not be accurate since an adversary could learn from previous tries and choose the $l$ values accordingly based on earlier observations. So, except for the first run, the adversary would not pick the $l$ values with a uniform distribution from the domain of $l$. In order to do a more precise analysis we need to model the adversary behavior. We discuss how to do this for a particular class of problems called segmented oracles in the following section.
3. SEGMENTED ORACLES

Segmented oracle side channels provide observations about “segments” of the secret. For example, a segmented oracle side channel can provide an observable value (such as execution time) that enables an adversary to determine if the first character of the secret value (for example a password) matches to the public value (the input provided by the adversary). In general, a segmented oracle provides a distinct value for each matched segment (such as the matching the first character in the password, matching the first 2 characters, the first 3 characters, etc.)

Note that for the function $F_1$ shown in Figure 1, execution time serves as a segmented oracle side channel. The function terminates the execution immediately if it determines that the first character of the secret does not match the input and the execution time increases linearly with the number of segments that match. I.e., by observing the execution time, an adversary can figure out how many characters of the secret match the public input. Hence, for the function $F_1$, execution time acts as a segmented oracle side channel.

The function $F_1$ is a particular instance of an early termination optimized equality comparison. It returns false as soon as it discovers a mismatch in order to avoid unnecessary comparisons. This is a common programming pattern found in many library functions which results in segmented oracle timing attacks [17, 18] that are remedied by implementing constant time functionally equivalent versions of those comparison functions that operate over sensitive data, for example $F_2$, in order to remove the timing side channel [19, 20, 21]. Our approach provides a method for automatically quantifying the amount of information an adversary can gain by a function under a segmented oracle side channel attack, indicating whether a constant time implementation is necessary.

Now, let us discuss the adversary model. We are assuming that the adversary is runs a function $F$ multiple times with different $l$ values (but the secret $h$ stays the same) and records the corresponding observables, while trying to figure out $h$. Further we assume that the analyzed programs are deterministic, i.e., given $h$ and $l$ values, $F(h, l)$ returns one observable value $o$ which represents the observations of the adversary after executing $F(h, l)$. For segmented oracles the observable values consist of a set of values $o \in \{o_0, o_1, \ldots, o_n\}$ where $o_0$ denotes no segments of the input $(l)$ and secret $(h)$ match, $o_i$ denotes $i$ segments of the secret match the input, and $o_n$ denotes the secret completely matches the input.

We call each execution of $F$ a run. So, the adversary is generating a sequence of $l$ values to run the program multiple times, the intuition being that each run reveals some new information about the secret. We can formalize the adversary as a function $A$ that takes all the prior history as input (which is a sequence of tuples where each tuple is a $l$ value and the corresponding observable for the execution of function $F$ with that $l$ value). Note that the $h$ value is constant and does not change from one execution to the other.

We can model the whole system $S = (A, F)$, where the adversary $A$ generates $l$ values for multiple executions of the function $F$ in order to determine the secret $h$, as follows.

Given the system $S = (A, F)$ we may want to compute the probability of determining the secret after $k$ runs, i.e., having $\text{seq} = k$ when $S$ terminates. Or, we may want to compute the information leakage (i.e., entropy) for $k$ runs.

One approach would be to analyze the system $S$ without restricting the adversary. However, this would take into account behaviors such as the adversary trying the same $l$ value over and over again even though it does not match the secret. When analyzing vulnerabilities of a software system, we have to focus on the behavior of the best adversary.

```plaintext
let H_k = - \sum_{o_1, o_2, \ldots, o_l} p(o_1, o_2, \ldots, o_l) \times \log_2(p(o_1, o_2, \ldots, o_l))
```

For the segmented oracles, it is easy to specify the best adversary $A_{\text{B}}$ [17]. This adversary works as follows: Let $(l^1, o^1), (l^2, o^2), \ldots, (l^k, o^k)$ be the run history. The adversary generates $l^{k+1}$ for the $k+1$st run as follows:

- If $o^k \neq o^{k-1}$ and $o^k = o_i$, then the adversary constructs $l^{k+1}$ as follows: $\forall j, 1 \leq j < i : l^{k+1}[j] = l^k[j]$ (part of $l$ that already matched remains the same), $l^{k+1}[i] \neq l^k[i]$, (use a different value for the first part that did not match in the last try) and rest of the $l^{k+1}$ is randomly generated.
- If $o^k = o^{k-1}$, then let $m$ be the smallest number where $o^m = o^k$ and let $o^{k-1} = o^m$. Then the adversary constructs the $l^{k+1}$ as follows: $\forall j, 1 \leq j < i : l^{k+1}[j] = l^k[j]$ (part of $l$ that already matched remains the same) and $\forall j, m \leq j < k : l^{k+1}[j] \neq l^k[j]$ (use a different value then the values that have already been tried for the first part that does not match) and rest of the $l^{k+1}$ is randomly generated.

Let $S_k$ denote the execution of the system $S = (A_{\text{B}}, F)$ where the function $F$ is executed $k$ times, i.e., $|\text{seq}| = k$. We can ask the following question: What is the probability of the adversary $A_{\text{B}}$ guessing the password in exactly $k$ tries?

Note that, execution of $S_k$ will generate observable sequences $o^1, o^2, \ldots, o^k$ where for all $1 \leq t \leq k$, $o^t = o_t$, $o^{t+1} = o_t \Rightarrow j \geq i$. I.e., since we are using the best adversary model $A_{\text{B}}$, the observable values in the sequence will be non-decreasing. The adversary will never produce a worse match than the one in the previous try. Another constraint for the observable sequences is that if $o_k$ appears in a sequence, then $o_k$ is the last observable of the sequence since $S$ terminates when $o_k$ is observed.

We can calculate the probability of determining the password in exactly $k$ tries as the probability of generating the observable sequences $o^1, o^2, \ldots, o^k$ where $l \leq k$, observable values in the sequence are non-decreasing, and $o_l^k = o_k$.

Let $p(o^1, o^2, \ldots, o^k)$ denote the probability of $S_k$ generating that particular observable sequence. Then we can compute the entropy for $S_k$ (i.e., the information leakage within the first $k$ runs) as follows:

$$H_k = - \sum_{o_1, o_2, \ldots, o_l} p(o_1, o_2, \ldots, o_l) \times \log_2(p(o_1, o_2, \ldots, o_l))$$

where $SEQ^k$ is the set of all non-decreasing observable sequences that can be generated by the first $k$ iterations of $S = (A_{\text{B}}, F)$. For every sequence $o^1, o^2, \ldots, o^l \in SEQ^k$: 1) $l \leq k$, 2) the observable values in the sequence are non-decreasing, 3) if $o_n$ appears in the sequence, then it is the last observable in the sequence, and 4) if $o_n$ does not appear in the sequence, then $l = k$.  

4. MULTI-RUN SIDE-CHANNEL ANALYSIS

In this section we present two approaches to multi-run analysis of segmented oracles. The first approach is intuitive and more general; it is applicable to any adversary model. However, this approach requires the probabilistic symbolic execution of an adversary model which executes the program multiple times, and thus it suffers from the path explosion problem. To address this problem, for the best adversary model, we propose a more scalable approach with a novel computation of the leakage which requires the probabilistic symbolic execution on only one run of the program.

4.1 Multi-Run Symbolic Execution

Our first approach for multi-run side-channel analysis is described with the following two steps. First, we create a model of the attack scenario, explained in Section 3 where an adversary can provide the low inputs, and execute the program a number of times. Then, we use probabilistic symbolic execution to explore all possible observations of the model and compute the probability for each observation. Shannon entropy and channel capacity of the leaks are easily derived from the probabilities.

In this work, a model in our analysis is a Java bytecode program, written as a driver for the program under test. Since the secret \( h \) and the inputs of the adversary \( l_1, \ldots, l_k \) are not known in advance, they are modeled by symbolic variables in symbolic execution. Without any constraints on \( l_1, \ldots, l_k \), this is a model for a very naive adversary, who repeatedly tries to guess the secret with random values, and learns nothing from the previous attempts.

To model an adversary who gains information through observing program executions and revises the input domain accordingly, we use the assume-guarantee reasoning in SE to impose the constraints on the inputs. We illustrate the approach by implementing a particular adversary model.

4.1.1 The best adversary model

Procedure \( S \) in Figure 4 depicts a driver modeling the best adversary described in section 3. Here an observation \( o' \) of the adversary indicates how many segments in the low input matched with the secret. The adversary is allowed to make \( k \) executions of \( F(h, l) \) but stops early if all the segments are matched, i.e. \( o' = |h| \). The instruction \assumee \( i \) assume \( (l'[s] \neq l'[s]) \) implemented by the built-in API Debug.assume in SPF, is used to impose constraints on the inputs.

The best adversary is characterized by two sets of assumptions. The first set of assumptions reflect the fact that, for the segment being searched \( s \), the best adversary selects an input different from the ones in the previous executions. When the adversary discovers new segments of the secret, i.e. when \( o' > o' - 1 \), it keeps these segments for the inputs of the following executions, and moves on to search for the next segment. This is modeled by the second set of assumptions in the procedure.

4.1.2 Computation of information leakage

In this approach, the computation of leakage does not depend on any particular adversary model \( S = (A,F) \), i.e. it can be applied to any model with any assumptions made by the adversary, or even no assumptions at all.

For our analysis, we extend classical symbolic execution to keep track of the assumptions ASM in a symbolic path. At a low level, ASM is implemented with exactly the same data structure as the path condition. When executing the instruction \assumee \( c \), symbolic execution updates the path condition \( PC = PC \land c \), and checks satisfiability with a constraint solver. Symbolic execution advances to the next instruction if the updated \( PC \) is satisfiable, and it backtracks otherwise. Our extension for symbolic execution updates \( ASM \leftarrow ASM \land c \) only when the updated \( PC \) is satisfiable. Thus, there is no constraint solving overhead for \( ASM \).

We perform symbolic execution, with our extension, on the model \( S = (A,F) \) to explore all possible observations. Each observation of \( S \) is a sequence of observations of \( F \): \( \overline{o} = (o_1, o_2, \ldots, o_k) \) where \( 1 \leq n \leq k \). For each \( \overline{o} \), we also obtain from symbolic execution the path condition \( PC \), that leads to that observation, and the assumptions ASM \( i \) on that path.

We denote by \( D_h, D_{l_1}, D_{l_2} \ldots D_{l_k} \) the domains of \( h, l_1, l_2 \ldots l_k \) respectively. The input space is then \( D = D_h \times D_{l_1} \times \cdots \times D_{l_k} \). If there is no assumptions on the low inputs, \( l_i \), can take any value \( D_i \). Hence, the search space of the adversary is \( D \), and the probability of observing \( \overline{o} \) is computed by

\[
p(\overline{o}) = \frac{|PC|}{|D|}
\]

In the case the adversary has some knowledge about the input, modeled by the assumptions, the revised domain of \( \overline{o} \) is \( |ASM_i| \), and hence its probability is

\[
p(\overline{o}) = \frac{|PC|}{|ASM_i|}
\]

Both \( |PC| \) and \( |ASM_i| \) are computed by model counting tools integrated in probabilistic symbolic execution. We will discuss in more details about these tools in later section.

4.2 Multi-Run Analysis Using Single-Run Symbolic Execution

As shown in the previous section, we are able to compute the probabilities of observation sequences by performing a complete symbolic execution of a program which simulates the adversary strategy of repeated guessing. However, performing a complete symbolic execution over all iterations of adversary behavior can become prohibitively expensive. Therefore, we seek to avoid this expensive computation. In
this section, we describe how to compute the sequence probabilities using symbolic execution and model counting from only a single iteration of the adversary strategy, by taking advantage of the segmented nature of observations which reveal the secret.

Notation. For a segmented oracle the low ($l$) and high ($h$) inputs are compared incrementally. The $n$ segments of $l$ and $h$ are denoted by $[l[0], \ldots, l[n-1]]$ and $[h[0], \ldots, h[n-1]]$, respectively. We write $h[j]:l[j]$ for the "slice" of $h$ from index $i$ to index $j$, and similarly for $l$. We let $D_i$ be the domain size of $l[i]$, or equivalently, the domain size of $h[i]$, and we write $D = \langle D_0, D_1, \ldots, D_{n-1} \rangle$ for the vector of these domain sizes. We will write $D_{i,j}$ to denote the subvector of $D$ of indices $i$ through $j$, and $\prod D$ for the product of all elements of $D$.

**Probability Computation.** By performing a symbolic execution of a single run of $F(h,l)$ we can automatically generate the set of observables and corresponding path conditions, $\{(o_i, PC_i) : 0 \leq i \leq n\}$. Without loss of generality we assume an order of observables, $o_0 < o_1 < \ldots < o_{n+1}$, and we assume that the path conditions are in the form given below, a generalization of the path conditions given in Section 2. Path constraints of this form result from symbolic execution of comparison functions which utilize the early termination optimization programming pattern, as described in Section 2.

$$PC_i = \begin{cases} 
(l[i] \neq h[i]) \land \left( \bigwedge_{j=0}^{i-1} h[j] = l[j] \right) & \text{if } i < n \\
\left( \bigwedge_{j=0}^{i-1} h[j] = l[j] \right) & \text{if } i = n
\end{cases}$$

Due to the segmented nature of the comparison between $l$ and $h$, we can consider the size of the domain $D_i$ for each segment, that is, the number of possible values to which each segment can be assigned, independently. Then each $PC_i$ determines a combinatorial restriction on the set of $D$.

- In the case of $PC_n$ where each $h[i] = l[i]$, we have that for any of the $D_i$ values for $l[i]$, the value of $h[i]$ is constrained to a single value. Therefore, the product of the domain sizes must be equal to $|PC_n|$.

- For $PC_i$ ($i < n$), we have that $h[j] = l[j]$ for $j < i$, and so for any of the $D_i$ values for $l[i]$, $h[i]$ is constrained to be a single value. Since, $h[i] \neq l[i]$, for any of the $D_i$ values for $l[i]$, there are $D_i - 1$ possible values for $h[i]$. Finally for $j > i$ there is no constraint on the relationship between $l[i]$ and $h[i]$ and so there are $D_i$ possible values for each of them.

The combinatorial argument above can be summarized by the following system of equations:

$$\prod D = |PC_n|$$

$$\prod D \cdot (w_i - 1) \cdot \prod D_{i+1,n-1} = |PC_i|$$

This system of equations can be solved for each $w_i$ via reverse substitution using the following recurrence:

$$D_i = \frac{|PC_i|}{|PC_n| \cdot \prod D_{i+1,n-1}} + 1$$

Once we have determined the domain sizes of the individual segments, we are in a position to compute the probability any particular observation sequence. Let $p(\vec{c}\mid \vec{D})$ be the probability of observation sequence $\vec{c}$ given a vector of segment domains $\vec{D}$. In addition, we define $D'_i$ to be the vector of domains constrained by $PC_i$. That is $D'_i = \langle 1, 1, \ldots, D_i - 1, D_i + 1, \ldots, D_{n-1} \rangle$. Then $p(\vec{c}\mid \vec{D})$ can be computed recursively using the following logic:

- Base Case: if $\vec{c} = \alpha_i$ is a sequence of length 1, the probability of $\alpha_i$ is $\prod D'_0 / \prod D$, that is, the number of remaining possible inputs that are consistent with $(\alpha_i, PC_i)$, out of the total number of inputs in the domain.

- Recursive Case: if $\vec{c} = (\alpha^1, \alpha^2, \ldots, \alpha^k)$ is a sequence of length $k$ we can think of it as $\alpha^1$ followed by a sequence of length $k-1$. Then computing $p(\vec{c}\mid \vec{D})$ reduces to computing the probabilities of $p(\alpha^1\mid D'_1)$ and $p(\alpha^2, \ldots, \alpha^k\mid D')$ and multiplying.

The above presented computation results in the same probabilities that are computed by a full probabilistic symbolic execution analysis of the adversary’s complete attack behavior. Given the probabilities, we can simply apply the entropy formula. We have implemented both methods and experimentally verified that they produce the same results. However, the second method is significantly faster. We discuss this in Section 6 containing our experimental results.

### 5. STRING CONSTRAINTS

In this section we discuss our extensions to SPF for string analysis and model counting. Our work is motivated by the extensive use of string manipulation in modern software applications. Some common reasons for using string manipulation are: 1) creation of documents in HTML or XML format, 2) runtime code generation, 3) creation of queries for backend databases, 4) validation and sanitization of user input.

In order to analyze modern software systems it is necessary to handle string constraints.

$$F \rightarrow C \mid \neg F \mid F \land F \mid F \lor F$$

$$C \rightarrow S = S$$

$$\mid \text{match}(S, S)$$

$$\mid \text{contains}(S, S)$$

$$\mid \text{begins}(S, S)$$

$$\mid \text{ends}(S, S)$$

$$\mid I = I \mid I < I$$

$$S \rightarrow v \mid *$$

$$\mid S, S \mid S \mid S^*$$

$$\mid \text{replace}(S, S, S)$$

$$\mid \text{substring}(S, I, I)$$

$$\mid \text{charAt}(I)$$

$$\mid \\text{String}(I)$$

$$I \rightarrow v \mid n$$

$$\mid I + I \mid I - I \mid I \ast n$$

$$\mid \text{length}(S)$$

$$\mid \text{indexOf}(S, S)$$

$$\mid \text{substring}(\text{int, int})$$

$$\mid \text{replace}(\text{String, String})$$
5.1 Symbolic Execution with Strings

We experimented with two approaches for handling of string operations in SPF: 1) Numeric encoding reduces string operations to numeric constraints, 2) String encoding maps string operations to string constraints.

All existing string solvers are limited in their capabilities of handling mixed integer and numeric constraints. Many of the current solutions to symbolic execution for strings support only a subset of such operations. The approaches to model counting are even more limited. We therefore implemented a numeric encoding approach in SPF that uses the low-level Java implementations of the String classes and uses models only for the native calls in these methods. This effectively reduces all string operations to low-level numeric operations over arrays of characters (representing the strings).

The low-level Java implementations of string operations can thus be analyzed with SPF and the generated numeric constraints can be handled with available solvers such as Z3 [31]. Furthermore, off-the-shelf numeric model-counting procedures for numeric constraints such as LattE [27] can be used.

This numeric encoding approach has the advantage that it is robust and general (it can handle arbitrary combinations of numeric and string constraints) but it can only analyze symbolic strings of fixed length.

The second approach we implemented maps string operations directly to string constraints. For this approach we built on SPF's existing capabilities for symbolic execution over strings [36]. SPF maintains an additional path condition that encodes directly operations from Java String, StringBuilder and StringBuffer APIs. The constraints maintained by SPF are built from string expressions described by the grammar in Figure 3.

In this string encoding approach, SPF does not analyze the implementations of the string operations. Instead it builds string expressions based on the string operations (and assumes the implementations are correct). For example, when if(cmd.indexOf(’ ’)==-1) is executed with symbolic value s1 for cmd, the method indexof is not actually executed inside SPF but rather a symbolic string expression is created which can later appear in the symbolic string expressions and path conditions built by the analysis, e.g. symbolic constraint s1.indexOf(’ ’)==-1 is added to the string PC. We integrated the model-counting string constraint solver ABC to SPF to support this string encoding approach.

5.1.1 Automata Based Constraint Solving

Automata Based Model Counter (ABC) is an automata based constraint solver that also supports model counting [3]. ABC was originally developed for string constraint solving. In this paper, in order to support model counting both for numeric and string constraints and their combinations, we extended ABC to support numeric constraints. Below we explain how ABC converts numeric and string constraints to automata.

String Constraints.

Given an automaton A, let \( \mathcal{L}(A) \) denote the set of strings accepted by A. Given a constraint F and a string variable v, our goal is to construct a deterministic finite automaton (DFA) \( \mathcal{A} \) such that \( \mathcal{L}(A) = [F, v] \) where \( [F, v] \) denotes the set of strings for which F evaluates to true when substituted for the variable v in F.

Figure 6: Automata (a) for the string constraint \( \neg(x \in (ab)^*) \land \text{LENGTH}(x) \geq 1 \) and (b) for the numeric constraint \( x - y < 1 \).

Let us define an automata constructor function \( \mathcal{A} \) such that, given a string constraint F and a variable v, \( \mathcal{A}(F, v) \) is an automaton where \( \mathcal{L}(\mathcal{A}(F, v)) = [F, v] \). Below we discuss how to implement the automata constructor function \( \mathcal{A} \).

Let us first discuss Boolean operators. Given a constraint \( \neg F \), in order to construct \( \mathcal{A}(\neg F, v) \) we can first construct \( \mathcal{A}(F, v) \) and use automata complement to construct \( \mathcal{A}(\neg F, v) \) where \( \mathcal{L}(\mathcal{A}(\neg F, v)) = \Sigma^* - \mathcal{L}(\mathcal{A}(F, v)) \). For constraints in the form \( F_1 \land F_2 \) and \( F_1 \lor F_2 \), we can first construct \( \mathcal{A}(F_1, v) \) and \( \mathcal{A}(F_2, v) \). Then we can construct \( \mathcal{A}(F_1 \land F_2, v) \) and \( \mathcal{A}(F_1 \lor F_2, v) \) using automata product, where \( \mathcal{L}(\mathcal{A}(F_1 \land F_2, v)) = \mathcal{L}(\mathcal{A}(F_1, v)) \cap \mathcal{L}(\mathcal{A}(F_2, v)) \) and \( \mathcal{L}(\mathcal{A}(F_1 \lor F_2, v)) = \mathcal{L}(\mathcal{A}(F_1, v)) \cup \mathcal{L}(\mathcal{A}(F_2, v)) \).

Automata constructor \( \mathcal{A}(C, v) \) for basic constraints C can be implemented for each basic constraint type shown in Figure 4 as discussed in [3]. As an example, consider the string constraint \( F \equiv \neg(x \in (ab)^*) \land \text{LENGTH}(x) \geq 1 \) over the alphabet \( \Sigma = \{a, b\} \). In order to construct \( \mathcal{A}(F, x) \), we first construct \( \mathcal{A}(x \in (ab)^*, x) \), and \( \mathcal{A}(\text{LENGTH}(x) \geq 1, x) \), and use automata complement and automata product operations to obtain the resulting automaton shown in Figure 5(a).

A constraint F may have more than one variable. In that case, we use the same algorithm describe above to construct an automaton for each variable in F. If two variables appear in the same basic constraint, we do a projection for each of them. In a multi-variable constraint, for each variable v, we would get an over-approximation of the truth-set \( \mathcal{A}(F, v) \supseteq [F, v] \). We can eliminate over-approximation by solving the constraint iteratively. At each iteration, we initialize each \( \mathcal{A}(F, v) \) to automaton that is obtained in the previous iteration for the same v. We stop the iteration when there is no more change in any \( \mathcal{A}(F, v) \). Note that, using multiple variables, one can specify constraints with non-regular truth sets. For example, given the constraint \( F \equiv x = y \cdot y \), \( [F, x] \) is not a regular set, so we cannot construct an automaton precisely recognizing its truth set. In that case, we put a bound on the number of iterations for constraint solver and return an over-approximation of the truth set when the bound is reached.

Numeric Constraints.

In order to handle numeric constraints in ABC, we implemented the automata construction techniques for linear
arithmetic constraints on integers [3]. The approach we use can handle arithmetic constraints that consist of linear equalities and inequalities (\(=, \neq, >, \geq, \leq, <\)) and logical operations (\(\land, \lor, \neg\)).

Similar to string constraints, the goal is to create an automaton that accepts solutions to the given formula. However, for numeric constraints, it is necessary to keep relationships between integer variables in order to preserve precision. For example, given a numeric constraint such as \(2x - y = 0\), we would like the automaton to recognize the tuples \((x, y)\) such that \((x, y) \in \{(0, 0), (1, 2), (2, 4), (3, 6), \ldots\}\). If we separate the constraints for values \(x\) and \(y\) and recognize the set \(0, 1, 2, 3, \ldots\) for \(x\) and the set \(0, 2, 4, 6, \ldots\) for \(y\), then we would get tuples such as \((2, 2)\), which are not allowed by the constraint \(2x - y = 0\). To address this, we use multi-track automata which is a generalization of finite state automata.

A multi-track automaton accepts tuples of values by reading one symbol from each track in each transition. I.e., given an alphabet \(\Sigma\), a \(k\)-track automaton has an alphabet \(\Sigma^k\).

For numeric constraints, we use the alphabet \(\Sigma = \{0, 1\}\). The numeric automata accept tuples of integer values in binary form, starting from the least significant digit.

We implement an automata constructor function \(\mathcal{A}\) for numeric constraints such that, given a numeric constraint \(F\), \(\mathcal{A}(F)\) is an automaton where \(L(\mathcal{A}(F)) = [F]\). Note that, for numeric constraints, \(\mathcal{A}(F)\) accepts tuples of integer values, one for each variable in the constraint \(F\). Each variable in \(F\) is mapped to a unique track of the multi-track automaton that we construct.

The automata constructor \(\mathcal{A}\) for numeric constraints handles the boolean operators \(\neg, \land, \lor\) the same way as the automata constructor for string constraints. Each basic numeric constraint is in the form \(\sum_{i=1}^{n} a_i \cdot x_i + a_0 = 0\), where \(a_0 \in \{-, \neq, >, \geq, \leq, <\}\), \(a_i\) denote integer coefficients and \(x_i\) denote integer variables. The automata construction for basic numeric constraints relies on a basic binary adder state machine construction [3]. The state machine starts from a state labeled with the constant term \(a_0\). It reads the first binary digit of all the variables, computes the result of the sum for the first digit and the carry. The next state is the state that corresponds to the new carry. Using each digit and the current carry, it is possible to compute the next carry which define the transitions of the state machine. Accepting states are determined based on the operation \(op\). For example, if the operation is \(=\), then all the resulting digits should be equal to 0 and the carry should also be 0. So the state 0 is accepting and all transitions that result in a non-zero digit go to the sink state. In order to handle negative values, 2’s-complement representation is used.

As an example, in Figure 3(b) we show the multi-track automaton that accepts tuples of integer values that satisfy the constraint \(x - y < 1\) (the transitions are labeled with the digit for variable \(x\) on top of the digit for variable \(y\)).

If a component \(c_i\) is a set of linear integer constraints, it is simplified and normalized further by using the Omega library [1]. Latte [27] is then used on this normalized constraint to count the models of \(c_i\).

As we described above, ABC is an automata-based constraint solver that, given a constraint \(F\) constructs an automaton \(\mathcal{A}(F)\) where \(L(\mathcal{A}(F)) = [F]\). Note that, \(|F| = |L(\mathcal{A}(F))|\). So, in order to count the number of solutions for a constraint \(F\), we need to count the number of strings accepted by \(\mathcal{A}(F)\). Counting the number of accepted strings by an automaton corresponds to counting the number of accepting paths [3]. For example, consider the automaton for constraint \(F \equiv -(x \in (ab)^* \land \text{LENGTH}(x) \geq 1)\) shown in Figure 5(a). In the language \(L(\mathcal{A}(F))\), we have zero strings of length 0 (\(\not\varepsilon \not\in L(\mathcal{A}(F))\)), two strings of length 1 \(\{(a, b)\}\), three strings of length 2 \(\{(aa, ba, bb)\}\), and so on.

Given an automaton \(A\), consider its corresponding language \(L(A)\). Let \(L_i(A) = \{w \in L(A) : |w| = i\}\), the language of strings in \(L(A)\) with length \(i\). Then \(L(A) = \bigcup_{i \geq 0} L_i(A)\). The cardinality of \(L(A)\) can be computed as \(|L_i(A)| = \sum_{i \geq 0} |L_i(A)|\).

Note that the number of strings accepted by an automaton could be infinite in the presence of loops. In applications of model counting (such as probabilistic symbolic execution) a model counting query is accompanied with a bound that limits the domain of the variable. For string variables this is the length of the strings, whereas for numeric variables it is the number of bits. These correspond to the length of the accepted strings for our automata representation of string and numeric constraints.

Computation of \(|L(A)|\) within a bound can be done by constructing the adjacency matrix of the automaton based on its transition relation, and then using matrix exponentiation to compute the number of accepting paths. We first add a new extra state to the automaton and connect each accepting state to this new state with \(\lambda\)-transitions where \(\lambda\) is a new padding symbol that is not in the alphabet of \(A\). The augmented DFA preserves both the language and count information of \(A\). From this augmented DFA we construct the adjacency matrix \(T\) where matrix entry \(T_{ij}\) corresponds to the number of transitions from state \(s_i\) to state \(s_j\). Let \(n + 1\) denote the new state that was added. We can compute \(|L_m(A)|\) by computing the matrix \(T^m\) by matrix exponentiation where \(L_m(A) = T_{n+1,n+1}^m\). Moreover, we can compute \(\sum_{m \leq m \leq m} |L_i(\lambda)|\) by modifying the matrix \(T\) to matrix by adding a self-loop to the new state that was added. After that modification \(\sum_{m \leq m \leq m} |L_i(\lambda)| = T_{n+1,n+1}^m\).

Note that this approach works both for numeric and string constraint automata. Hence, using an automata-based constraint solver provides a general approach to model counting.

6. EXPERIMENTS

To validate the effectiveness of our methods, we first evaluated ABC by comparing it with Latte. Next we compare the efficiency of using multi-run vs. single-run symbolic execution for computing the entropy after a sequence of observations. Lastly, we have tested our side-channel analysis on: 1) the password checking function described in Section 3 which is susceptible to a timing attack, and 2) a compression function which contains a side channel based on the size of the compressed output file.
computing the entropy after the adversary makes

6.2 Single- and Multi-run Symbolic Execution

As described in Section 4, we have given two methods for computing the entropy after the adversary makes \( k \) observations: performing symbolic execution over the \( k \)-composition of the program under an adversary model (Section 4.1.1) and performing symbolic execution over a single copy of the program and then using mathematical formula to infer the multi-run entropies (Section 4.2).

We ran both analyses on the password checking example and, as expected, we see in Figure 7 that the multi-run analysis takes much longer, due to the exploration of many more paths generated by symbolic execution.

6.3 Password Checker

We also present results on the timing analysis of the password checking function. We present results only for multi-run analysis using single-run execution here, as we have just described that it is much faster and produces the same results. We first describe results for a small configuration where we fix the alphabet size to 4 and the password length to 3. We assume that the adversary can make \( k \) guesses, and we compute the remaining entropy and the information leakage as shown in Figure 8. There are \( 4^3 = 64 \) possible inputs for \( h \) giving \( \log_2 64 = 6 \) bits for the initial entropy. As the adversary makes more guesses, the remaining entropy decreases from 6 to 0. Indeed, our analysis shows that the entropy is 0 for \( k \geq 10 \). Symmetrically, we can see that the information leakage increases with more guesses, from 0 to 6, indicating that all information about the secret is leaked after 10 guesses. Thus, we conclude that the adversary needs at most 10 guesses to fully determine the secret.

We also analyzed a larger configuration. For a password of length 10 and an alphabet of size 128, we incrementally increased the guessing budget of the adversary and determined that 15 guesses are required to reveal 1 bit of information. This analysis took 135.34 seconds.

6.4 Text Concatenation and Compression

We further analyzed side channels that depend on the size of the output. One example of such an attack is known as “Compression Ratio Info-leak Made Easy” (CRIME) [37].

The function `concatSAndCompress()` shown in Figure 9 accepts an input `low` which is controlled by the adversary, concatenates it with a secret high value `high`, and then uses the Lempel-Ziv (LZ77) [44] compression algorithm on the resulting string. We do not show the code for `LZ77compress` here, as it requires approximately 60 lines of Java code.

```java
public concatAndCompress (String low){
    return LZ77compress(high.concat(low));
}
```

Figure 10: A function with a size-based side channel.
The basic idea behind the attack is that if the adversary provides a value for low that does not have a common prefix with high, then there will be little compression. However, if low and high do share a prefix, this will result in a higher compression ratio. This is real-world vulnerability that can be used to reveal secret web session tokens to a malicious user by observing compressed network packet size.\footnote{The work by considering probabilistic systems to account for secrets that change over time.} Such a user is able to control input through, say, a web form, which is later concatenated with session information and sent to the server. For instance, suppose the secret value is the text sessionkey:xb5da. If the malicious user sets the value of low to be the text string sessionkey:abcde, he will observe less compression than if he sets low to be sessionkey:xb5da. In this way, the attacker is able to make repeated guesses and incrementally learn more information about prefixes of the secret. Thus, the $\text{concatAndCompress()}$ function acts as a segmented oracle with a side channel on the size of the output.

We apply our analysis to $\text{concatAndCompress()}$ and we are able to compute the information leakage for a given budget on the number of guesses used by the adversary. Due to the complexity of the LZ77 algorithm, symbolic execution becomes more expensive than in the case of the password checking function. For a secret of length 3 and alphabet size 4, single-run symbolic execution generates 187 path conditions leading to 4 different observables. For each observable $o_i$, we built the disjunction of all the PCs that result in $o_i$ and used Z3 to prove logical equivalence to the PC formulation in Section 4.2. Using the single-run method we then determined that the $\text{concatAndCompress()}$ function leaks all information about the secret after 10 executions by the adversary. Using ABC Numeric for model counting, the total running time of this analysis is 8,695 seconds. We repeated this experiment using ABC String as the model counter. The same result took 152,332 seconds to compute, due to the complex nature of the string operations contained in the LZ77 compression algorithm.

7. RELATED WORK

Side-channels attacks received significant attention in previous work.\footnote{A malicious user by observing compressed network packet size.} Kocher\footnote{An earlier version of ABC was presented in\cite{ABC-1}.} addresses timing attacks against cryptographic systems using statistical techniques treating the attack as a signal detection problem where the signal consists of the timing variation due to the target secret bit and “noise” results from measurement inaccuracies and timing variations in the unknown secret bits. Brumley and Boneh\footnote{Further LeakWatch\cite{LeakWatch} estimates leakage in Java program based on sampling program executions on concrete inputs. Köpf and Basin\cite{Kopf1} present a multi-run analysis based on an enumeration algorithm. Mardziel et al.\cite{Mardziel1} generalizes the work by considering probabilistic systems to account for secrets that change over time.} further study timing attacks against OpenSSL implementations and show how to extract private keys using similar testing techniques over multiple rounds of attacks on OpenSSL-based web servers running on a machine on a local network. CacheAudit\footnote{Further LeakWatch\cite{LeakWatch} estimates leakage in Java program based on sampling program executions on concrete inputs. Köpf and Basin\cite{Kopf1} present a multi-run analysis based on an enumeration algorithm. Mardziel et al.\cite{Mardziel1} generalizes the work by considering probabilistic systems to account for secrets that change over time.} uses static analysis for cache side channels to derive formal, quantitative security guarantees for a comprehensive set of side-channel adversaries, based on observing cache states, traces of hits and misses, and execution times.

Quantitative measurement of information leakage has been an active area of research. Early work\footnote{In previous work\cite{6, 25, 7, 10}, Kocher\cite{Kopf1} addresses timing attacks against cryptographic systems using statistical techniques treating the attack as a signal detection problem where the signal consists of the timing variation due to the target secret bit and “noise” results from measurement inaccuracies and timing variations in the unknown secret bits. Brumley and Boneh\cite{ABC-1} further study timing attacks against OpenSSL implementations and show how to extract private keys using similar testing techniques over multiple rounds of attacks on OpenSSL-based web servers running on a machine on a local network. CacheAudit\cite{ABC-2} uses static analysis for cache side channels to derive formal, quantitative security guarantees for a comprehensive set of side-channel adversaries, based on observing cache states, traces of hits and misses, and execution times.} measured the number of tainted bits, not an information-theoretic bound. Most previous work\footnote{In previous work\cite{6, 25, 7, 10}, Kocher\cite{Kopf1} addresses timing attacks against cryptographic systems using statistical techniques treating the attack as a signal detection problem where the signal consists of the timing variation due to the target secret bit and “noise” results from measurement inaccuracies and timing variations in the unknown secret bits. Brumley and Boneh\cite{ABC-1} further study timing attacks against OpenSSL implementations and show how to extract private keys using similar testing techniques over multiple rounds of attacks on OpenSSL-based web servers running on a machine on a local network. CacheAudit\cite{ABC-2} uses static analysis for cache side channels to derive formal, quantitative security guarantees for a comprehensive set of side-channel adversaries, based on observing cache states, traces of hits and misses, and execution times.} quantify the leakage in one run of the program given a concrete value of low input. Single-run analysis is addressed in\cite{14} where bounded model checking is used over the $k$-composition of a program to determine if it can yield $k$ different outputs. Further LeakWatch\cite{LeakWatch} estimates leakage in Java program based on sampling program executions on concrete inputs. Köpf and Basin\cite{Kopf1} present a multi-run analysis based on an enumeration algorithm. Mardziel et al.\cite{Mardziel1} generalizes the work by considering probabilistic systems to account for secrets that change over time.}

8. CONCLUSIONS

We presented a symbolic execution approach for side channel analysis. We illustrated our approach on side channels with segmented oracles and we gave an efficient computation of leakage over multiple attack steps. Our technique leverages satisfiability checking and model counting over complex constraints involving both string and numeric operations. In the future we plan to extend our side-channel analysis with segmented oracles in the presence of noisy observations.

Acknowledgments

We thank the anonymous reviewers for their constructive comments. This material is based on research sponsored by NSF under grants CCF-1548848 and CCF-1549161 and 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. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government. The term “Segmented Oracle” was defined by DARPA in a white paper distributed to us.
9. REFERENCES


