String Analysis for Side Channels with Segmented Oracles

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

•Software and its engineering → Formal software verification; •Security and privacy → Logic and verification;

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 [27, 3] 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 [36].

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

DOI: 10.1145/1235

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 [19] 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 [44]. 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 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 F2 (char[] h, char[] l){
  matched = true;
  for (int i = 0; i < h.length; i++) {
    if (h[i] != l[i]) matched = false;
    else matched = matched;
  }
  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., $o \in \{o_0, o_1, \ldots o_n\}$, 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

digit in the PIN, then secod digit etc. using the side-channel information). The question is: can we formalize the amount of information leaked and can we automatically compute it?

2.1 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:

- $PC_0 \equiv h[0] \neq l[0]$
- $PC_1 \equiv h[0] = l[0] \wedge h[1] \neq l[1]$
- $PC_2 \equiv h[0] = l[0] \wedge h[1] = l[1] \wedge h[2] \neq l[2]$

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

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 charter 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 $p(o_i) = |PC_{o_i}|/|D|$ where the probability of the input value completely matching the password h is $p(o_n)$.

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:

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

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.

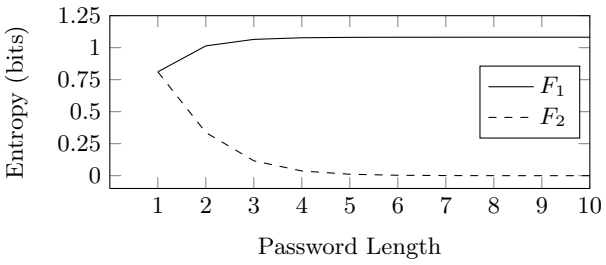


Figure 3: Entropy after a single guess for functions F_1 and F_2, for password length ranging from 1 to 10.

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:

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

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

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 [**?**, **?**, **?**, **?**, **?**]. These vulnerabilities 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 [**?**, **?**, **?**, **?**]. Our approach provides a method for automatically quantifying the amount of information an advesary 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 is 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 $|seq| = k$ 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.

procedure $S = (A, F)$, initially $seq \leftarrow nil$
  repeat
    $l \leftarrow A(seq)$
    $o \leftarrow F(h, l)$
    $seq \leftarrow append(seq, \langle l, o \rangle)$
  until $(o = o_n)$

For the segmented oracles, it is easy to specify the best adversary A_B [17]. This adversary works as follows: Let $\langle l^1, o^1 \rangle, \langle l^2, o^2 \rangle, \ldots, \langle l^k, o^k \rangle$ 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 $o^k = o^m = o_i$, 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 $\forall j, m \leq j < k \ : \ l^{k+1}[i] \neq l^j[i]$ (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 $S = (A_B, F)$ where the function F is executed k times, i.e., $|seq| = k$. We can ask the following question: What is the probability of the adversary A_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 $1 \leq t \leq k, o^t = o_i \wedge o^{t+1} = o_j \Rightarrow j \geq i$. I.e., since we are using the best adversary model A_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_n appears in a sequence, then o_n is the last observable of the sequence since S terminates when o_n is observed.

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

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:

$$\mathcal{H}(S^k) = -\sum_{o^1, o^2, \ldots, o^l \in SEQ^k} p(o^1, o^2, \ldots, o^l) \times \log_2(p(o^1, o^2, \ldots, o^l))$$

where SEQ^k is the set of all non-decreasing observable sequences that can be generated by the first k iterations of $S = (A_B, F)$. 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 $l^1, \dots l^k$ are not known in advance, they are modeled by symbolic variables in symbolic execution. Without any constraints on $l^1, \dots l^k$, 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 **??** depicts a driver modeling the best adversary described in section 3. Here an observation o^i 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. $o^i = |h|$. The instruction ***assume***, 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 search s, the best adversary selects an input different from the ones in the previous executions. When the adversary discovers more segments of the secret, i.e. when $o^i > o^{i-1}$, she 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 ***assume***(c), symbolic execution updates the path condition $PC \leftarrow PC \wedge c$, 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 $ASM \leftarrow ASM \wedge c$ only when the updated PC is satisfiable. Thus, there is no constraint solving overhead for ASM.

```
procedure  S = (A_B, F)
vars
  s: the current segment of h being searched
  b: the first time s is searched
  o^0, o^1, ... o^k: observations of the adversary
begin
  s ← 1, b ← 1, o^0 ← 0
  for all i ∈ [1..k] {
    for all j ∈ [b..i) { assume (l^i[s] ≠ l^j[s]) }
    o^i ← F(h, l^i)
    if (o^i = |h|) { return }
    if (o^i > o^{i-1}) {
      for all j ∈ [i+1..k] {
        for all n ∈ [s..o^i] { assume (l^j[n] = l^i[n]) }
      }
      s ← o^i + 1, b ← i + 1
    }
  }
end
```

Figure 4: Adversary Model

We performs 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: $\overrightarrow{o_i} = \langle o^1, o^2 \dots o^n \rangle$ where $1 \leq n \leq k$. For each $\overrightarrow{o_i}$, we also obtain from symbolic execution the path condition PC_i that leads to that observation, and the assumptions ASM_i on that path.

We denote by $D_h, D_1, D_2 \dots D_k$ the domains of $h, l_1, l_2 \dots l_k$ respectively. The input space is then $D = D_h \times D_1 \times \cdots \times D_k$. 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 $\overrightarrow{o_i}$ is computed by

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

In the case the adversary has some knowledge about the input, modeled by the assumptions, the revised domain of $\overrightarrow{o_i}$ is $|ASM_i|$, and hence its probability is

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

Both $|PC_i|$ 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 $h[i:j]$ 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 $\mathbf{D} = \langle D_0, D_1, \ldots, D_{n-1} \rangle$ for the vector of these domain sizes. We will write $\mathbf{D}_{i:j}$ to denote the subvector of $\mathbf{D}$ of indices i through j, and $\prod \mathbf{D}$ for the product of all elements of $\mathbf{D}$.

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.

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

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 $\mathbf{D}$.

- 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 $l[j]$, $h[j]$ 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:

$$\begin{cases} \prod \mathbf{D} = |PC_n| \\ \prod \mathbf{D} \cdot (w_i - 1) \cdot \prod \mathbf{D}_{i+1:n-1} = |PC_i| \end{cases}$$

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

$$D_i = \frac{|PC_i|}{|PC_n| \cdot \prod \mathbf{D}_{i+1:n-1}} + 1$$

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 $p(\overrightarrow{o}|\mathbf{D})$ be the probability of observation sequence $\overrightarrow{o}$ given a vector of segment domains $\mathbf{D}$. In addition, we define $\mathbf{D}'_i$ to be the vector of domains constrained by PC_i. That is $\mathbf{D}'_i = \langle 1, 1, \ldots, D_i - 1, D_i + 1, \ldots, D_n \rangle$. Then $p(\overrightarrow{o}|\mathbf{D})$ can be computed recursively using the following logic:

- Base Case: if $\overrightarrow{o} = o_i$ is a sequence of length 1, the probability of o_i is $(\prod \mathbf{D}'_i) / (\prod \mathbf{D})$, that is, the number of remaining possible inputs that are consistent with (o_i, PC_i), out of the total number of inputs in the domain.
- Recursive Case: if $\overrightarrow{o} = \langle o^1, o^2, \ldots o^k \rangle$ is a sequence of length k we can think of it as o^1 followed by a sequence of length $k-1$. Then computing $p(\overrightarrow{o}|\mathbf{D})$ reduces to computing the probabilities of $p(o^1|\mathbf{D}'_i)$ and $p(\langle o^2, \ldots, o^k \rangle|\mathbf{D}'_i)$ 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.

$$\begin{array}{rcll} F & \rightarrow & C \mid \neg F \mid F \wedge F \mid F \vee F & (1) \\ C & \rightarrow & S = S & (2) \\ & \mid & \textsc{match}(S, S) & (3) \\ & \mid & \textsc{contains}(S, S) & (4) \\ & \mid & \textsc{begins}(S, S) & (5) \\ & \mid & \textsc{ends}(S, S) & (6) \\ & \mid & I = I \mid I < I & (7) \\ S & \rightarrow & v \mid s & (8) \\ & \mid & S.S \mid S + S \mid S^* & (9) \\ & \mid & \textsc{replace}(S, S, S) & (10) \\ & \mid & \textsc{substring}(S, I, I) & (11) \\ & \mid & \textsc{charAt}(I) & (12) \\ & \mid & \textsc{toString}(I) & (13) \\ I & \rightarrow & v \mid n & (14) \\ & \mid & I + I \mid I - I \mid I * n & (15) \\ & \mid & \textsc{length}(S) & (16) \\ & \mid & \textsc{indexOp}(S, S) & (17) \end{array}$$

Figure 5: String Constraints

We define the set of string constraints using the grammar shown in Figure 4 where C denotes the basic constraints, n denotes integer values, $s \in \Sigma^*$ denotes string values, and v denotes string and integer variables. This constraint language can model complex string operations available in Java and in many modern programming languages such as `boolean matches(String)`, `int indexOf(String, int)`, `String substring(int, int)`, `String replace(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 4.

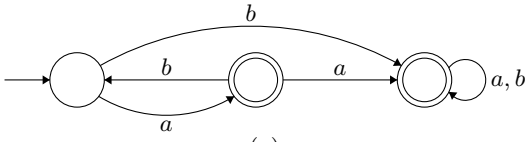
In this string encoding approach, SPF does not analyze the implementations of the string operations. Instead it builds string expressions based on the string operations (and assumes the implementations are correct). For example, when `if(cmd.indexOf(' ')==-1)` is executed with symbolic value `s1` for `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

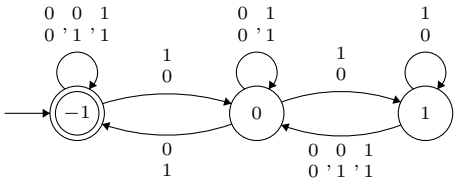
Automata Based model Counter (ABC) is an automata based constraint solver that also supports model counting [3]. ABC was originally developed for string constraint solving. In this paper, in order to support model counting both for numeric and string constraints and their combinations, we extended ABC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



(a)



(b)

Figure 6: Automata (a) for the string constraint $\neg(x \in (ab)^*) \wedge \text{LENGTH}(x) \geq 1$ 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 $F_1 \wedge F_2$ and $F_1 \vee F_2$, we can first construct $\mathcal{A}(F_1, v)$ and $\mathcal{A}(F_2, v)$. Then we can construct $\mathcal{A}(F_1 \wedge F_2, v)$ and $\mathcal{A}(F_1 \vee F_2, v)$ using automata product, where $\mathcal{L}(\mathcal{A}(F_1 \wedge F_2, v)) = \mathcal{L}(\mathcal{A}(F_1, v)) \cap \mathcal{L}(\mathcal{A}(F_2, v))$ and $\mathcal{L}(\mathcal{A}(F_1 \vee F_2, v)) = \mathcal{L}(\mathcal{A}(F_1, v)) \cup \mathcal{L}(\mathcal{A}(F_2, v))$.

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 $F \equiv \neg(x \in (ab)^*) \wedge \text{LENGTH}(x) \geq 1$ 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 $F \equiv x = y \ . \ y$, $[\![F, x]\!]$ 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 [5]. The approach we use can handle arithmetic constraints that consist of linear equalities and inequalities ($=, \neq, >, \geq, \leq, <$) and logical operations ($\wedge, \vee, \neg$).

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 set of values for 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 Σ, a k-track automaton has an alphabet Σ^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 $\mathcal{L}(\mathcal{A}(F)) = [\![F]\!]$. 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 $\neg, \wedge, \vee$ the same way as the automata constructor for string constraints. Each basic numeric constraint is in the form $\sum_{i=1}^{n} a_i \cdot x_i + a_0 \; op \; 0$, where $op \in \{=, \neq, >, \geq, \leq, <\}$, 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 [5]. 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 5(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).

5.2 Model Counting

Model counting for numeric path conditions using Latte has been implemented in our previous work [12]. As model counting is expensive we perform several optimizations. First the path condition PC is partitioned into independent components which can be solved separately: $PC = c_1 \wedge c_2 \wedge \cdots \wedge c_n$, where a variable x in a component c_i does not appear in any other component. Therefore $|PC| = \Pi_i |c_i|$.

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, give a constraint F constructs an automaton $\mathcal{A}(F)$ where $\mathcal{L}(\mathcal{A}(F)) = [\![F]\!]$. Note that, $|F| = |\mathcal{L}(\mathcal{A}(F))|$. 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 $F \equiv \neg(x \in (ab)^*) \wedge \textsc{length}(x) \geq 1$ shown in Figure 5(a). In the language $\mathcal{L}(\mathcal{A}(F))$, we have zero strings of length 0 ($\varepsilon \notin \mathcal{L}(\mathcal{A}(F))$), 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 $\mathcal{L}(A)$. Let $\mathcal{L}_i(A) = \{w \in \mathcal{L}(A) : |w| = i\}$, the language of strings in $\mathcal{L}(A)$ with length i. Then $\mathcal{L}(A) = \bigcup_{i \geq 0} \mathcal{L}_i(A)$. The cardinality of $\mathcal{L}(A)$ can be computed as $|\mathcal{L}(A)| = \sum_{i \geq 0} |\mathcal{L}_i(A)|$.

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 $|\mathcal{L}(A)|$ 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 λ-transitions where λ 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 $T_{i,j}$ 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 $|\mathcal{L}_m(A)|$ by computing the matrix T^m by matrix exponentiation where $|\mathcal{L}_m(A)| = T^m_{n+1,n+1}$. Moreover, we can compute $\sum_{0 \leq i \leq m} |\mathcal{L}_i(A)|$ my modifying the matrix T to matrix by adding a self-loop to the new state that was added. After that modification $\sum_{0 \leq i \leq m} |\mathcal{L}_i(A)| = T^m_{n+1,n+1}$.

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.

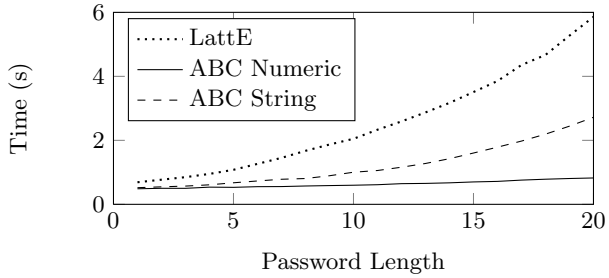


Figure 7: Time comparison for computing single guess entropy using ABC and LattE.

6.1 Timing Performance of Model Counting

Symbolic PathFinder already contained an implementation of path constraint model counting using LattE [11]. In addition, we integrated ABC in SPF for counting solutions to path constraints. Our experiments show that ABC and LattE produce identical model counting results. To compare running time, we analyzed the password checking function. We compare the end-to-end running time of performing symbolic execution, collecting path constraints, and performing model counting on all generated constraints in order to compute the information leakage of a single run by the adversary. We find that the implementation using ABC is significantly faster than the previous implementation that uses Latte. As shown in Figure 6, for a fixed alphabet size of 4, we see that the running time increases with the password length for both ABC and LattE, and that the ABC Numeric implementation is significantly faster, with ABC String second fastest, and the Latte implementation slowest.

However, we do not claim that ABC is faster than Latte. ABC is implemented as a shared library in SPF allowing for direct function calls to the model counter. On the other hand, in order for SPF to pass constraints to Latte, they are first preprocessed and simplified using the Omega library and then saved to a set of files. Latte is then invoked on these files and the model counts are parsed back into JPF. In order to make any claims about the relative efficiency of ABC and LattE we will need to do a comparison of the constraint model counting capabilities directly. This is future work.

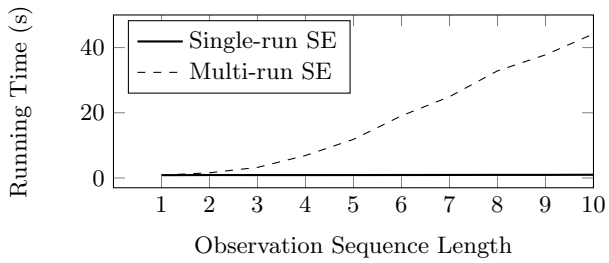


Figure 8: Time for multi-run and single-run SE.

The remaining experiments were conducted using ABC Numeric as the model counter, due to the relative execution speed of the implementation within SPF.

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.

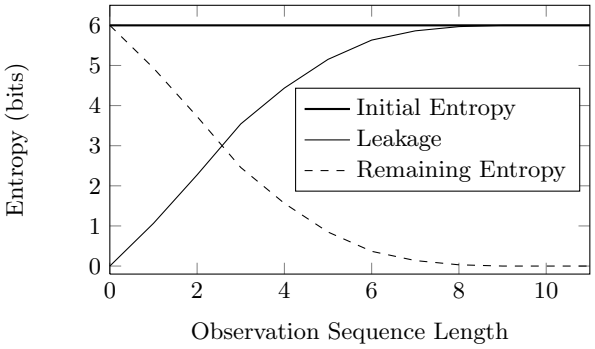


Figure 9: Information leakage and remaining entropy for password checking function.

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.

```
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 [19]. 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 `high` is the text `sessionkey:xb5du`. 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 `concatAndCompress()` function acts as a segmented oracle with a side channel on the size of the output.

We apply our analysis to `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 we used Z3 to prove logical equivalence to the PC formulation in Section 4.2. Using the single-run method we then determined that the `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 results 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 [6, 23, 7, 10]. Kocher [23] 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 [6] 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 [10] 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 are of research. Early work [30] measured the number of tainted bits, not an information-theoretic bound. Most previous work [9, 4, 34, 22, 32, 33], quantify the leakage in one run of the program given a concrete value of low input. Single-run analysis is addressed in [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 [8] estimates leakage in Java program based on sampling program executions on concrete inputs. Köpf and Basin [24] present a multi-run analysis based on an enumeration algorithm. Mardziel et al. [29] generalizes the work by considering probabilistic systems to account for secrets that change over time.

In previous work [35] we give a formulation of multi-run side channel analysis using symbolic execution and MaxSMT solving. The focus of that work is to synthesize the worst case attack for arbitrary side channels, in the context of non-adaptive attacks. In contrast here we assume the worst case attack is known and it is adaptive, i.e. the attacker changes the public input based on the observations made so far. Further we give an efficient computation of leakage tailored to side channels with segment oracles for string manipulating programs.

There has been significant amount of work on string constraint solving [15, 20, 16, 38, 13, 43, 25, 2, 26, 39]; however none of these solvers provide model-counting functionality. SMC is the only other model-counting string constraint solver that we are aware of [28]. ABC is strictly more precise than SMC. Namely SMC cannot propagate string values across logical connectives which reduces its precision during model counting, whereas we can handle logical connectives without losing precision. We can also handle complex string operations such as `replace` that SMC cannot handle. ABC builds on the automata-based string analysis tool Stranger [42, 40, 41] determined to be the best string solver in terms of precision and efficiency in a recent empirical study [18]. An earlier version of ABC was presented in [3]. In this paper we extended the functionality of ABC to handle all string operations in Java, numeric constraints, and the ability to perform model counting for numeric constraints. Further, we integrated ABC in SPF.

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

[1] Omega. http://www.cs.umd.edu/projects/omega/.

[2] P. A. Abdulla, M. F. Atig, Y. Chen, L. Holík, A. Rezine, P. Rümmer, and J. Stenman. String constraints for verification. In *Proceedings of the 26th International Conference on Computer Aided Verification (CAV)*, pages 150–166, 2014.

[3] A. Aydin, L. Bang, and T. Bultan. Automata-based model counting for string constraints. In *Proceedings of the 27th International Conference on Computer Aided Verification (CAV)*, pages 255–272, 2015.

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

[5] C. Bartzis and T. Bultan. Efficient symbolic representations for arithmetic constraints in verification. *Int. J. Found. Comput. Sci.*, 14(4):605–624, 2003.

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

[7] S. Chen, R. Wang, X. Wang, and K. Zhang. Side-channel leaks in web applications: A reality today, a challenge tomorrow. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 191–206, Washington, DC, USA, 2010. IEEE Computer Society.

[8] T. Chothia, Y. Kawamoto, and C. Novakovic. Leakwatch: Estimating information leakage from java programs. In M. Kutylowski and J. Vaidya, editors, *Computer Security - ESORICS 2014 - 19th European Symposium on Research in Computer Security, Wroclaw, Poland, September 7-11, 2014. Proceedings, Part II*, volume 8713 of *Lecture Notes in Computer Science*, pages 219–236. Springer, 2014.

[9] D. Clark, S. Hunt, and P. Malacaria. A static analysis for quantifying information flow in a simple imperative language. *J. Comput. Secur.*, 15(3):321–371, Aug. 2007.

[10] G. Doychev, B. Köpf, L. Mauborgne, and J. Reineke. Cacheaudit: A tool for the static analysis of cache side channels. *ACM Trans. Inf. Syst. Secur.*, 18(1):4, 2015.

[11] A. Filieri, C. S. Pasareanu, and W. Visser. Reliability analysis in symbolic pathfinder. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, pages 622–631, 2013.

[12] A. Filieri, C. S. Păsăreanu, and W. Visser. Reliability analysis in symbolic pathfinder. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 622–631, Piscataway, NJ, USA, 2013. IEEE Press.

[13] V. Ganesh, M. Minnes, A. Solar-Lezama, and M. C. Rinard. Word equations with length constraints: What's decidable? In *Proceedings of the 8th International Haifa Verification Conference (HVC)*, pages 209–226, 2012.

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

[15] P. Hooimeijer and W. Weimer. A decision procedure for subset constraints over regular languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 188–198, 2009.

[16] P. Hooimeijer and W. Weimer. Solving string constraints lazily. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 377–386, 2010.

[17] M. Joye. Basics of side-channel analysis. In *Cryptographic Engineering*, chapter 13, pages 367–382. 2009.

[18] S. Kausler and E. Sherman. Evaluation of string constraint solvers in the context of symbolic execution. In *Proceedings of the 29th ACM/IEEE International Conference on Automated software engineering (ASE)*, pages 259–270, 2014.

[19] J. Kelsey. Compression and information leakage of plaintext. In *Fast Software Encryption, 9th International Workshop, FSE 2002, Leuven, Belgium, February 4-6, 2002, Revised Papers*, pages 263–276, 2002.

[20] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. Hampi: a solver for string constraints. In *Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA)*, pages 105–116, 2009.

[21] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

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

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

[24] B. Köpf and D. A. Basin. An information-theoretic model for adaptive side-channel attacks. In P. Ning, S. D. C. di Vimercati, and P. F. Syverson, editors, *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007*, pages 286–296. ACM, 2007.

[25] G. Li and I. Ghosh. PASS: string solving with parameterized array and interval automaton. In *Proceedings of the 9th International Haifa Verification Conference (HVC)*, pages 15–31, 2013.

[26] T. Liang, A. Reynolds, C. Tinelli, C. Barrett, and M. Deters. A DPLL(T) theory solver for a theory of strings and regular expressions. In *Proceedings of the 26th International Conference on Computer Aided Verification (CAV)*, pages 646–662, 2014.

[27] J. A. D. Loera, R. Hemmecke, J. Tauzer, and R. Yoshida. Effective lattice point counting in rational convex polytopes. *Journal of Symbolic Computation*,

38(4):1273 – 1302, 2004. Symbolic Computation in Algebra and Geometry.

[28] L. Luu, S. Shinde, P. Saxena, and B. Demsky. A model counter for constraints over unbounded strings. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, page 57, 2014.

[29] P. Mardziel, M. S. Alvim, M. W. Hicks, and M. R. Clarkson. Quantifying information flow for dynamic secrets. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 540–555, 2014.

[30] S. McCamant and M. D. Ernst. Quantitative information flow as network flow capacity. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 193–205, New York, NY, USA, 2008. ACM.

[31] Microsoft Inc. Z3 SMT Solver. http://z3.codeplex.com.

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

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

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

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

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

[37] J. Rizzo and T. Duong. The crime attack. Ekoparty Security Conference, 2012.

[38] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for javascript. In *Proceedings of the 31st IEEE Symposium on Security and Privacy*, 2010.

[39] M. Trinh, D. Chu, and J. Jaffar. S3: A symbolic string solver for vulnerability detection in web applications. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1232–1243, 2014.

[40] F. Yu, M. Alkhalaf, and T. Bultan. Stranger: An automata-based string analysis tool for php. In *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 154–157, 2010.

[41] F. Yu, M. Alkhalaf, T. Bultan, and O. H. Ibarra. Automata-based symbolic string analysis for vulnerability detection. *Formal Methods in System Design*, 44(1):44–70, 2014.

[42] F. Yu, T. Bultan, M. Cova, and O. H. Ibarra. Symbolic string verification: An automata-based approach. In *Proceedings of the 15th International SPIN Workshop on Model Checking Software (SPIN)*, pages 306–324, 2008.

[43] Y. Zheng, X. Zhang, and V. Ganesh. Z3-str: A z3-based string solver for web application analysis. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 114–124, 2013.

[44] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, May 1977.