Attack Synthesis for Strings using Meta-Heuristics

Seemanta Saha[1], Ismet Burak Kadron[1], William Eiers[1], Lucas Bang[2], and Tevfik Bultan[1]

[1] University of California, Santa Barbara, {seemantasaha,kadron,weiers,bultan}@cs.ucsb.edu

[2] Harvey Mudd College, bang@cs.hmc.edu

ABSTRACT

Information leaks are a significant problem in modern computer systems and string manipulation is prevalent in modern software. We present techniques for automated synthesis of side-channel attacks that recover secret string values based on timing observations on string manipulating code. Our attack synthesis techniques iteratively generate inputs which, when fed to code that accesses the secret, reveal partial information about the secret based on the timing observations, leading to recovery of the secret at the end of the attack sequence. We use symbolic execution to extract path constraints, automata-based model counting to estimate the probability of execution paths, and meta-heuristic methods to maximize information gain based on entropy for synthesizing adaptive attack steps.

1. INTRODUCTION

Modern software systems store and manipulate sensitive information. It is crucial for software developers to write code in a manner that prevents disclosure of sensitive information to arbitrary users. However, computation that accesses sensitive information can have attacker-measurable characteristics that leaks information. This can allow a malicious user to infer secret information by measuring characteristics such as execution time, memory usage, or network delay. This type of unintended leakage of secret information due to non-functional behavior of a program is called a side-channel vulnerability. In this paper, we focus on side-channel vulnerabilities that result from timing characteristics of string manipulating functions. For a given function F that performs computation over strings, we automatically synthesize a side-channel attack against F. The synthesized attack consists of a sequence of inputs that a malicious user can use to leak information about the secret by observing timing behavior. By synthesizing an attack, we provide a proof of vulnerability for the function.

Our approach uses symbolic execution to extract constraints characterizing the relationship between secret strings in the program, attacker controlled inputs, and side-channel observations.We compare several methods for selecting the next attack input based on meta-heuristics for maximizing the amount of information gained.

Our contributions in this paper can be summarized as follows: (1) to the best of our knowledge, this is the first work which performs attack synthesis specifically targeting side-channels in string-manipulating programs; (2) we provide and experimentally compare several approaches to attack synthesis for strings. We make use of meta-heuristics for searching the input space, including model-based searching, random searching, simulated annealing, and genetic algorithms; and (3) we present attack synthesis techniques based on automata-based model counting.

*This material is based on research supported by an Amazon Research Award and by DARPA under the 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.

```
public Boolean checkPIN(String h, String l){
  for (int i = 0; i < 4; i++)
    if (h.charAt(i) != l.charAt(i))
      return false;
  return true;
}
```

Figure 1: PIN checking example.

A Motivating Example. Consider a PIN-based authentication function (Fig. 1) with inputs: 1) a secret PIN h, and 2) a user input, l. Both h and l are strings of digit characters ("0"–"9") of length 4. We have adopted the nomenclature used in security literature where h denotes the *high-security* value (the secret PIN) and l denotes the *low-security* value, (the input that the function compares with the PIN). The function compares the PIN and the user input character by character and returns `false` as soon as it finds a mismatch. Otherwise it returns `true`.

One can infer information about the secret h by measuring the execution time. For each length of the common prefix of h and l, the execution time will differ. Notice that if h and l have no common prefix, then `checkPIN` will have the shortest execution time since the loop body will be executed only once; this corresponds to 63 Java bytecode instructions. If h and l have a common prefix of one character, we see a longer execution time since the loop body executes twice (78 instructions). In the case that h and l match completely, `checkPIN` has the longest running time (108 instructions). There are 5 observable values since there are 5 different execution times proportional to the length of the common prefix of h and l. Hence, an attacker can choose inputs and use the side-channel observations to determine how much of a prefix has matched. For this function, we automatically generated the constraints that characterize the length of the matching prefix and corresponding execution costs (number of executed bytecode instructions) using symbolic execution (Table 1). Our technique uses these constraints to synthesize an attack which determines the value of the secret PIN. We make use of an uncertainty function, based on Shannon entropy, to measure the progress of an attack (Section 3). Intuitively, the attacker's uncertainty, $\mathcal{H}$ starts off at some positive value and decreases during the attack. When $\mathcal{H} = 0$, the attacker has fully learned the secret (Table 2).

Table 1: Observation constraints generated by symbolic execution of the function in Figure 1.

i	Observation Constraint, ψ_i	o
1	$charat(l,0) \neq charat(h,0)$	63
2	$charat(l,0) = charat(h,0) \wedge charat(l,1) \neq charat(h,1)$	78
3	$charat(l,0) = charat(h,0) \wedge charat(l,1) = charat(h,1) \wedge$ $charat(l,2) \neq charat(h,2)$	93
4	$charat(l,0) = charat(h,0) \wedge charat(l,1) = charat(h,1) \wedge$ $charat(l,2) = charat(h,2) \wedge charat(l,3) \neq charat(h,3)$	108
5	$charat(l,0) = charat(h,0) \wedge charat(l,1) = charat(h,1) \wedge$ $charat(l,2) = charat(h,2) \wedge charat(l,3) = charat(h,3)$	123

Table 2: Attack inputs (l), uncertainty about the secret ($\mathcal{H}$), and observations (o). Prefix matches are shown in **bold**.

Step	$\mathcal{H}$	l	o	Step	$\mathcal{H}$	l	o
1	13.13	"8299"	63	15	5.906	"**13**92"	93
2	12.96	"0002"	63	16	5.643	"**131**6"	93
3	9.813	"**1**058"	78	17	5.321	"**130**8"	93
4	9.643	"**1**477"	78	18	4.906	"**136**2"	93
5	9.451	"**1**583"	78	19	4.321	"**137**8"	93
6	9.228	"**1**164"	78	20	3.169	"**1338**"	108
7	8.965	"**1**950"	78	21	3.000	"**1332**"	108
8	8.643	"**1**220"	78	22	2.807	"**1334**"	108
9	8.228	"**1**786"	78	23	2.584	"**1333**"	108
10	7.643	"**1**817"	78	24	2.321	"**1330**"	108
11	6.643	"**1**664"	78	25	2.000	"**1335**"	108
12	6.491	"**13**42"	93	26	1.584	"**1336**"	108
13	6.321	"**13**28"	93	27	0.000	"**1337**"	123
14	6.129	"**13**86"	93				

Suppose that the secret is "1337". The initial uncertainty is $\log_2 10^4 = 13.13$ bits of information. Our attack synthesizer generated input "8229" at the first step and makes an observation with cost 63, which corresponds to ψ_1. This indicates that $charat(h, 0) \neq 8$. Similarly, a second synthesized input, "0002", implies $charat(h, 0) \neq 0$ and the uncertainty is again reduced. At the third step the synthesized input "1058" yields an observation of cost 78. Hence, ψ_2 is the correct path constraint to update our constraints on h, which becomes:

$$charat(h, 0) \neq 8 \wedge charat(h, 0) \neq 0 \wedge charat(h, 0) = 1 \wedge charat(h, 1) \neq 0$$

We continue synthesizing inputs and learning constraints on h, which tell us more information about the prefixes of h, until the secret is known after 27 steps. At the final step, we make an observation which corresponds to ψ_5 indicating a full match and the remaining uncertainty is 0. In general, our search for attack inputs should drive the entropy to 0, so we propose entropy optimization techniques. This particular type of attack is a *segment attack* which is known to be a serious source of security flaws [2, 13, 17]. Our approach automatically synthesizes this attack.

2. AUTOMATIC ATTACK SYNTHESIS

In this section we give a two phase approach that synthesizes attacks (Procedure 1). We consider functions F that take as input a secret string $h \in \mathbb{H}$ and an attacker-controlled string $l \in \mathbb{L}$ and that have side-channel observations $o \in \mathbb{O}$.

Procedure 1 SynthesizeAttack($F(h, l), C_h, h^*$)
This procedure calls the GenerateConstraints and RunAttack functions to synthesize adaptive attacks.

1: $\Psi \leftarrow$ GenerateConstraints($F(h, l)$)
2: RunAttack($F(h, l), \Psi, C_h, h^*$)

Static Analysis Phase. The first phase generates constraints from the program for h, l, and o (Procedure 2). We perform symbolic execution on the program under test with the secret (h) and the attacker controlled input (l) marked as symbolic [10, 16]. Symbolic execution runs F on symbolic rather than concrete inputs resulting in a set of path constraints Φ. Each $\phi \in \Phi$ is a logical formula that characterizes the set of inputs that execute some path in F. During symbolic execution, we keep track of the side-channel observation for each path. As in other works in this area, we model the execution time of the function by the number of instructions executed [2, 14, 15]. 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. We augment symbolic execution to return a function *obs* that maps a path constraint ϕ to an observation o. Since an attacker cannot extract information from program paths that have indistinguishable side-channel observations, we combine observationally similar path constraints via disjunction (Procedure 2, line 4), where we say that $o \sim o'$ if $|o - o'| < \delta$ for a given threshold δ. The resulting *observation constraints* (denoted ψ_o and Ψ) characterize the relationship between the secret (h) the attacker input (l) and side-channel observations (o).

Attack Synthesis Phase. The second phase synthesizes a sequence of inputs that allow an attacker to incrementally learn the secret (Procedure 3). During this phase, we fix a secret h^*, unknown to the attacker. We maintain a constraint C_h on the possible values of the secret h. Initially, C_h merely specifies the domain of the secret. We call procedure AttackInput, which uses one of several entropy-based heuristics (Section 4), to determine the input value l^* for the current attack step. Then, the observation o that corresponds to running the program under attack with h^* and l^* is revealed. We update C_h to reflect the new constraint on h implied by the attack input and observation—we instantiate the corresponding observation constraint, $\psi_o[l \mapsto l^*]$, and conjoin it with the current C_h (line 5). Based on C_h, we compute an uncertainty measure for h at every step using Shannon entropy [7], denoted $\mathcal{H}$ (Section 3). The goal is to generate inputs which drive $\mathcal{H}$ as close as possible to zero, in which case there is no uncertainty and the secret is fully known. This attack synthesis phase repeats until it is not possible to reduce the uncertainty, $\mathcal{H}$, any further.

Procedure 2 GenerateConstraints($F(h, l)$)
Performs symbolic execution on function F with secret string h and attacker-controlled string l. The resulting path constraints are combined according to indistinguishability of observations.

1: $\Psi \leftarrow \emptyset$
2: $(\Phi, \mathcal{O}, obs) \leftarrow$ SymbolicExecution($F(h, l)$)
3: **for** $o \in \mathcal{O}$ **do**
4: $\quad \psi_o \leftarrow \bigvee_{\phi \in \Phi : obs(\phi) \sim o} \phi$
5: $\quad \Psi \leftarrow \Psi \cup \{\psi_o\}$
6: **return** Ψ

Procedure 3 RunAttack($F(h, l), \Psi, C_h, h^*$)
Synthesizes a sequence of attack inputs, l^*, for $F(h, l)$, given observation constraints Ψ, initial constraints on h (C_h), and unknown secret h^*. Function AttackInput is one of the variations described in Section 4.

1: $\mathcal{H} \leftarrow$ Entropy(C_h)
2: **while** $\mathcal{H} > 0$ **do**
3: $\quad l^* \leftarrow$ AttackInput(C_h, Ψ)
4: $\quad o \leftarrow F(h^*, l^*)$
5: $\quad C_h \leftarrow C_h \wedge \phi_o[l \mapsto l^*]$
6: $\quad \mathcal{H} \leftarrow$ Entropy(C_h)

3. ENTROPY-BASED OBJECTIVE FUNCTION

Here we derive an objective function to measure the amount of information an attacker expects to gain by choosing an input value l_{val} to be used in the attack search heuristics of Section 4. In the following discussion, H, L, and O are random variables representing high-security input, low-security input, and side-channel observation respectively. We use entropy-based metrics from the theory of quantitative information flow [18]. Given probability function $p(h)$, the *information entropy* of H, denoted $\mathcal{H}(H)$, which we interpret as the observer's *uncertainty*, is

$$\mathcal{H}(H) = -\sum_{h \in \mathbb{H}} p(h) \log_2 p(h) \tag{1}$$

Given conditional distributions $p(h|o, l)$, and $p(o|l)$ we quantify the attacker's expected *updated uncertainty* about h, given a candidate choice of $L = l_{val}$, with the expectation taken over all pos-

sible observations, $o \in O$. We compute the *conditional entropy of* H *given* O *with* $L = l_{val}$ as

$$\mathcal{H}(H|O, L = l_{val}) = -\sum_{o \in \mathbb{O}} p(o|l_{val}) \sum_{h \in \mathbb{H}} p(h|o, l_{val}) \log_2 p(h|o, l_{val}) \tag{2}$$

Now, we can compute the expected amount of information *gained* about h by observing o after providing input value l_{val}. The *mutual information* between H and O, given $L = l_{val}$ denoted $\mathcal{I}(H; O|L = l_{val})$ is the difference between the initial entropy of H and the conditional entropy of H given O when $L = l_{val}$:

$$\mathcal{I}(H; O|L = l_{val}) = \mathcal{H}(H) - \mathcal{H}(H|O, L = l_{val}) \tag{3}$$

Equation (3) is our objective function. Providing input $l_{val} = l^*$ which maximizes $\mathcal{I}(H; O|L = l_{val})$ maximizes information gained about h. Equations (1) and (2) rely on $p(h)$, $p(o|l)$, and $p(h|o, l)$, which may change at every step of the attack. Recall that during the attack, we maintain a constraint on the secret, C_h. Assuming that all secrets that are consistent with C_h are equally likely, at each step, we can compute the required probabilities using model counting. Given a formula F, performing model counting on F gives the number of satisfying solutions for F, which we denote $\#F$. Thus, we observe that $p(h) = 1/\#C_h$ if h satisfies C_h, otherwise 0. Hence, Equation (1) reduces to $\mathcal{H}(H) = \log_2(\#C_h)$.

Procedure 2 gives side-channel observations $\mathcal{O} = \{o_1, \ldots, o_n\}$ and constraints over h and l corresponding to each o_i, $\Psi = \{\psi_1, \ldots, \psi_n\}$. The probability that h takes on a value, constrained by a particular ψ_i, for a given l_{val} can be computed by instantiating ψ_i with l_{val} and then model counting. Thus, $p(h|o_i, l_{val}) = 1/\#\psi_i[l \mapsto l_{val}]$. Similarly, $p(o|l_{val}) = \#\psi_i[l \mapsto l_{val}]/\#C_h[l \mapsto l_{val}]$.

In this paper, the ENTROPY (Equation (1)) and MUTUALINFO (Equation (3)) functions refer to the appropriate entropy-based computation just described, where $p(h)$, $p(o|l)$, and $p(h|o, l)$ are computed using the MODELCOUNT procedure. We implement the MODELCOUNT procedure using the Automata-Based Model Counter (ABC) tool, which is a constraint solver for string and integer constraints with model counting capabilities [1].

4. ATTACK SYNTHESIS HEURISTICS

At every attack step the goal is to choose a low input l^* that reveals information about h^*. Here we describe different techniques for synthesizing attack inputs l^*. Each approach uses a different heuristic to explore a subset of the possible low inputs. To search the input space efficiently, we first observe that we need to restrict the search to those l that are consistent with C_h.

Constraint-based Model Generation. Our attack synthesis algorithm maintains a constraint C_h which captures all h values that are consistent with the observations so far (Procedure 3, line 5). Using the observation constraints Ψ (which identify the relation among the secret h, public input l and the observation o), we project C_h to a constraint on the input l, which we call C_l, and we restrict our search on l to the set of values allowed by C_l. I.e., we only look for l values that are consistent with what we know about h (which is characterized by C_h) with respect to Ψ. This approach is implemented in GETINPUT and GETNEIGHBORINPUT functions. To evaluate different heuristics, in our experiments we used either GETINPUT which returns an l_{val} or GETNEIGHBORINPUT which returns an l_{val} by mutating the previous l_{val}. These two functions are further classified as Restricted (R), in which only models of C_l are generated, or non-restricted (NR), in which we do not enforce l_{val} to be a model of C_l. For Procedures 4, 5, and 6, we can use either the restricted or non-restricted versions of GETINPUT and GETNEIGHBORINPUT.

Procedure 4 ATTACKINPUT-RA(C_h, Ψ)
Generates a low input at each attack step via random sampling.

1: $\mathcal{I} \leftarrow 0$
2: **for** i **from** 1 **to** K **do**
3: $\quad l_{val} \leftarrow$ GETINPUT(Ψ, C_h)
4: $\quad \mathcal{I}_{new} \leftarrow$ MUTUALINFO(Ψ, C_h, l_{val})
5: $\quad$ **if** $\mathcal{I}_{new} > \mathcal{I}$ **then**
6: $\quad\quad \mathcal{I} \leftarrow \mathcal{I}_{new}, l^* \leftarrow l_{val}$
7: **return** l^*

Search via Random Model Generation. As a base-line search heuristic, we make use of the approach described above for generating low values that are consistent with C_h. The simplest approach is to generate a single random model from C_l and use it as the next attack input. We call this approach Model-based (M). A slightly more sophisticated approach (Procedure 4) is to generate K random samples using C_l, compute the expected information gain for each of them using Equation (3) and choose the best one. We call this approach the Random Restricted (RA-R) heuristic (since it is restricted to models consistent with C_l, and hence C_h).

Procedure 5 ATTACKINPUT-SA(C_h, Ψ)
Generates a low input at each attack step via simulated annealing.

1: $t \leftarrow t_0$, $l_{val} \leftarrow$ GETINPUT(Ψ, C_h), $\mathcal{I} \leftarrow$ MUTUALINFO(Ψ, C_h, l_{val})
2: **while** $t \geq t_{min}$ **do**
3: $\quad l_{val} \leftarrow$ GETNEIGHBORINPUT(l_{val}, Ψ, C_h)
4: $\quad \mathcal{I}_{new} \leftarrow$ MUTUALINFO(Ψ, C_h, l_{val})
5: $\quad$ **if** $(\mathcal{I}_{new} > \mathcal{I}) \vee \left(e^{(\mathcal{I}_{new} - \mathcal{I})/t} > \text{RANDOMREAL}(0, 1)\right)$ **then**
6: $\quad\quad \mathcal{I} \leftarrow \mathcal{I}_{new}, l^* \leftarrow l_{val}$
7: $\quad t \leftarrow t - (t \times k)$
8: **return** l^*

Simulated Annealing. Simulated annealing (SA) is a meta-heuristic for optimizing an objective function $g(s)$ [11]. SA is initialized with a candidate solution s_0. At step i, SA chooses a neighbor, s_i, of candidate s_{i-1}. If s_i is an improvement, i.e., $g(s_i) > g(s_{i-1})$, then s_i is used as the candidate for the next iteration. If s_i is not an improvement ($g(s_i) \leq g(s_{i-1})$), then s_i is still used as the candidate for the next iteration with a small probability p calculated using the second part of disjunction at line 5 in Procedure 5. Intuitively, SA is a controlled random search that allows a search path to escape local optima by permitting the search to sometimes accept worse solutions. The acceptance probability p decreases over time, which is modeled using a search "temperature" which "cools off" and converges to a steady state. Our SA based approach is shown in Procedure 5 where we use GETNEIGHBORINPUT function to get new candidates.

Genetic Algorithm. A genetic algorithm (GA) searches for an optimal input to an objective function $g(s)$ by iteratively simulating a *population* of candidate solutions $P_i = \{s_1, \ldots s_n\}$ [9]. Each s_i is modeled as a set of *genes*. Here, a gene sequence consists a string's characters. At step i, we compute $g(s_j)$ as the *fitness* of each candidate. A new population P_{i+1} of *offspring candidates* is generated from P_i by selecting pairs (s, s') and performing genetic crossover and mutation and selecting top N candidates from P_i by fitness. Our GA-based approach is shown in Procedure 6.

Since GA applies mutation and crossover to generate new values, restricted model generation does not apply. To restrict the search to l values that are consistent with C_l would require implementing

mutation and crossover operations with respect to C_l. We are not aware of a general approach for doing this, so during GA-based search, mutation and crossover operations can generate low values that are inconsistent with C_l (and hence C_h). Such values will have no information gain and will be ignored during search, but they can increase the search space and slow down the search.

Procedure 6 AttackInput-GA(C_h, Ψ)
Generates low input at each attack step using a genetic algorithm.

```
1: I ← 0
2: for j from 1 to popSize do
3:     pop[j] ← GetInput(Ψ, C_h)
4: for i from 1 to K do
5:     for j from 1 to popSize do
6:         popFit[j] ← MutualInfo(Ψ, C_h, pop[j])
7:         if popFit[j] > I then
8:             I ← popFit[j],  l* ← pop[j]
9:     cand ← MutateAndCrossover(pop, popFit, N)
10:    pop ← Append(SelectBest(pop, N), cand)
11: return l*
```

5. IMPLEMENTATIONS AND EXPERIMENTS

Implementation. We implemented Procedure 2 using Symbolic Path Finder (SPF) [16]. We implemented Procedure 3 as a Java program that takes the observation constraints generated by Procedure 2 as input, along with C_h and h^*. The variations of AttackInput from Section 4 (Procedures 4, 5, and 6) are implemented in Java. We implemented GetInput, GetNeighborInput, and ModelCount by extending the existing string model counting tool ABC. We added these features directly into the C++ source code of ABC along with corresponding Java APIs.

Benchmark Details. To evaluate the effectiveness of our attack synthesis techniques, we experimented on a benchmark of 8 string-manipulating programs utilizing various string operations, for different string lengths (Table 3). The functions `passCheckInsec` and `passCheckSec` are password checking implementations. Both compare a user input and secret password but early termination optimization (as described in the introduction) induces a timing side channel for the first one and the latter is a constant-time implementation. We analyzed the `stringEquals` method from the Java String library which is known to contain a timing side channel [8]. We discovered a similar timing side channel in `indexOf` method from the Java String library. Function `editDistance` example is an implementation of the standard dynamic programming algorithm to calculate minimum edit distance of two strings. Function `compress` is a basic compression algorithm which collapses repeated substrings within two strings. `stringInequality` and `stringCharInequality` functions check lexicographic inequality ($<, \geq$) of two strings whereas first one directly compares the strings and later compares characters in the strings.

Table 3: Benchmark details with the number of path constraints ($|\Phi|$) and the number of merged observation constraints ($|\Psi|$).

Benchmark	ID	Operations	Low Length	High Length	$\|\Phi\|$	$\|\Psi\|$
passCheckInsec	PCI	charAt,length	4	4	5	5
passCheckSec	PCS	charAt,length	4	4	5	1
stringEquals	SE	charAt,length	4	4	9	9
stringInequality	SI	$<, \geq$	4	4	2	2
stringCharInequality	SCI	charAt,length,$<, \geq$	4	4	80	2
indexOf	IO	charAt,length	1	8	9	9
compress	CO	begins,substring,length	4	4	5	5
editDistance	ED	charAt,length	4	4	2170	22

Experimental Setup. For all experiments, we use a desktop machine with an Intel Core i5-2400S 2.50 GHz CPU and 32 GB of DDR3 RAM running Ubuntu 16.04, with a Linux 4.4.0-81 64-bit kernel. We used the OpenJDK 64-bit Java VM, build 1.8.0 171. We ran each experiment for 5 randomly chosen secrets. We present the mean values of the results in Tables 4. For RA, we set the sample size K to 20. For SA, we set the temperature range (t to t_{min}) from 10 to 0.001 and cooling rate k as 0.1. For GA, we set population size *popSize* to 20, offspring size as 10, number of best selections N as 10.

Results. In this discussion, we describe the quality of a synthesized attack according to these metrics: the number of attack steps and overall change in entropy from $\mathcal{H}_{init}$ to $\mathcal{H}_{final}$. Attacks that do not reduce the final entropy to zero are called *incomplete*.

For all benchmarks, we compare 5 approaches: (1) model-based (M), (2) non-restricted random (RA-NR), (3) restricted random (RA-R), and (4) restricted simulated annealing (SA R), (5) restricted Genetic Algorithm (GA R). When we compare RA-NR and RA-R we observe that RA-NR is not as efficient as reducing the entropy because attack input generation fails to find any informative inputs for most of the steps. By restricting the input generation to consistent models using C_l as described in Section 4, we synthesize better attacks. Results on non-restricted and restricted versions of SA and GA were similar. We observe that the model-based technique (M), which also uses C_l to restrict the search space is faster than other techniques, as it greedily uses a single random model generated by ABC as the next attack input, with no time required to evaluate the objective function. M quickly generates attacks for most of the functions. We further examined those functions and determined that their objective functions are "flat" with respect to l. Any l_{val} that is a model for C_l at the current step yields the same expected information gain.

Table 4: Experimental results for model-based (M), random non-restricted (RA NR) and restricted (RA R), simulated-annealing restricted (SA R) and genetic algorithm restricted (GA R) heuristics. Time bound is set as 3600 seconds.

ID	$\mathcal{H}_{init}$	Metrics	M	RA NR	RA R	SA R	GA R
PCI	18.8	Time (s)	15.9	3600.0	3600.0	3600.0	3600.0
		Steps	54.2	110.0	39.4	34.5	41.5
		$\mathcal{H}_{final}$	0.0	9.3	5.7	8.4	8.5
PCS	18.8	Time (s)	3600.0	3600.0	3600.0	3600.0	3600.0
		Steps	118.0	42.5	41.4	33.2	38.0
		$\mathcal{H}_{final}$	18.8	18.8	18.8	18.8	18.8
SE	18.8	Time (s)	22.0	3600.0	3600.0	3600.0	3600.0
		Steps	62.2	85.0	42.6	25.3	30.8
		$\mathcal{H}_{final}$	0.0	11.8	6.1	11.1	8.4
SI	18.8	Time (s)	6.1	3600.0	78.3	268.2	218.5
		Steps	38.2	171.0	18.6	17.5	18.2
		$\mathcal{H}_{final}$	0.0	6.5	0.0	0.0	0.0
SCI	18.8	Time (s)	3600.0	3600.0	3600.0	3600.0	3600.0
		Steps	34.6	5.5	4.0	2.0	2.0
		$\mathcal{H}_{final}$	12.9	16.8	16.2	17.7	17.5
IO	37.6	Time (s)	29.1	3600.0	3600.0	3600.0	3600.0
		Steps	26.0	21.5	18.0	9.5	11.4
		$\mathcal{H}_{final}$	1.0	1.24	8.7	16.6	20.1
CO	18.8	Time (s)	3600.0	3600.0	3600.0	3600.0	3600.0
		Steps	734.0	183.0	147.0	83.0	97.8
		$\mathcal{H}_{final}$	13.48	7.9	9.2	10.3	9.1
ED	18.8	Time (s)	3600.0	3600.0	3600.0	3600.0	3600.0
		Steps	27.6	1.0	1.0	1.0	1.0
		$\mathcal{H}_{final}$	12.6	18.4	17.8	17.8	17.8

Although M is fast and generates attacks for each benchmark, experimental results show that it requires more attack steps compared (in terms of information gain) to the attacks generated by meta-heuristic techniques that optimize the objective function. As the experimental results show for the `stringInequality` example, a meta-heuristic technique can reduce $\mathcal{H}_{final}$ further but with fewer attack steps compared to the model-based approach (M). And, this case would be true for any example where different inputs at a specific attack step have different information

gain. Our experimental results also show the differences between random search (RA) and meta-heuristics (SA and GA). For the `stringInequality` example, SA is better than RA and GA. RA tries a random set of models consistent with C_l as low values, and picks the one with maximum information gain; GA uses random models consistent with C_l as the initial population and generates more low values using mutation and crossover of characters in the candidate strings; SA selects the first candidate as a random model consistent with C_l and then mutates the string to get other low values. Although GA builds the initial population using low values that are consistent with C_l, mutation and crossover operations can lead to low values which are not consistent with C_l. On the other hand, low values explored by SA and RA are always consistent with C_l, giving better results overall. Finally, we observe that some of our selected benchmarks are more secure against our attack synthesizer than others. In particular, `passCheckSec`, a constant-time implementation of password checking, did not leak any information through the side channel. Two other examples from the benchmark, `stringCharInequality` `editDistance` also did not succumb to our approach easily, due to the relatively large number of generated constraints 80 and 2170 respectively, indicating a much more complex relationship between the inputs and observations. To summarize, our experiments indicate that our attack synthesis approach is able to construct side-channel attacks against string manipulating programs, providing evidence of vulnerability (e.g. `passCheckInsec`). Further, when our attack synthesizer fails to generate attack steps (`passCheckSec`), or is only able to extract a relatively small information after many steps or significant computation time (`editDistance`), it provides evidence that the function under test is comparatively safer against side-channel attacks.

6. RELATED WORK

There has been prior work on analyzing side-channels [2, 4, 5, 15]. There has been recent results on synthesizing attacks or quantifying information leakage under a model where the attacker can make multiple runs of the system [2, 3, 6, 12, 15]. For example, LeakWatch [6] estimates leakage in Java programs based on sampling program executions on concrete inputs and Köpf et. al. [12] give a multi-run analysis based on an enumerative algorithm. There has also been prior work on quantifying information leakage using symbolic execution and model-counting techniques for integer constraints [3, 14, 15]. There are two previous results closely related to our work. The first [2] focuses on quantifying information flow through side channels for string-manipulating programs, applies only for programs that have a particular form of vulnerability known as segment oracle side-channels, and quantifies the amount of information leakage (does not synthesize attacks). The second [14] synthesizes side-channel attacks using either entropy-based or SAT-based objective functions, but works only for constraints in the theories of integer arithmetic or bit-vectors using model counters and constraint solvers for those theories.

7. CONCLUSION

In this paper we presented techniques for synthesizing adaptive attacks for string manipulating programs. To the best of our knowledge this is the first work which is able to automatically discover side channel vulnerabilities by synthesizing attacks targeting string manipulating functions. We presented several heuristics for attack synthesis and extended an existing automata-based model counter for attack synthesis. We experimentally demonstrated the effectiveness of our approach and compared several variations of attack-input selection heuristics.

8. REFERENCES

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

[2] Lucas Bang, Abdulbaki Aydin, Quoc-Sang Phan, Corina S. Pasareanu, and Tevfik Bultan. String analysis for side channels with segmented oracles. In *Proceedings of the 24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, 2016.

[3] Lucas Bang, Nicolas Rosner, and Tevfik Bultan. Online synthesis of adaptive side-channel attacks based on noisy observations. In *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P)*, 2018.

[4] David Brumley and Dan 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.

[5] Shuo Chen, Rui Wang, XiaoFeng Wang, and Kehuan 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.

[6] Tom Chothia, Yusuke Kawamoto, and Chris Novakovic. Leakwatch: Estimating information leakage from java programs. In *Computer Security - ESORICS 2014 - 19th European Symposium on Research in Computer Security, Wroclaw, Poland, September 7-11, 2014. Proceedings, Part II*, pages 219–236, 2014.

[7] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory (Wiley Series in Telecommunications and Signal Processing)*. Wiley-Interscience, 2006.

[8] Joel Sandin Daniel Mayer. Time trial: Racing towards practical remote timing attacks. `https://www.nccgroup.trust/globalassets/our-research/us/whitepapers/TimeTrial.pdf`, 2014.

[9] David E Goldberg. Genetic algorithms in search, optimization, and machine learning. Technical report, 1989.

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

[11] Scott Kirkpatrick, C Daniel Gelatt, and Mario P Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[12] Boris Köpf and David A. Basin. An information-theoretic model for adaptive side-channel attacks. In Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson, editors, *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007*, pages 286–296. ACM, 2007.

[13] Taylor Nelson. Widespread timing vulnerabilities in openid implementations. `http://lists.openid.net/pipermail/openid-security/2010-July/001156.html`, 2010.

[14] Quoc-Sang Phan, Lucas Bang, Corina S. Pasareanu, Pasquale Malacaria, and Tevfik Bultan. Synthesis of adaptive side-channel attacks. In *30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA*, 2017.

[15] Corina S. Păsăreanu, Quoc-Sang Phan, and Pasquale 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.

[16] Corina S. Păsăreanu, Willem Visser, David Bushnell, Jaco Geldenhuys, Peter Mehlitz, and Neha Rungta. Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis. *Automated Software Engineering*, pages 1–35, 2013.

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

[18] Geoffrey Smith. On the foundations of quantitative information flow. In *Proceedings of the 12th International Conference on Foundations of Software Science and Computational Structures (FOSSACS)*, 2009.