Incremental Attack Synthesis

Seemanta Saha*, William Eiers*, Ismet Burak Kadron*, Lucas Bang†, and Tevfik Bultan*

*University of California, Santa Barbara, CA 93106, {seemantasaha,weiers,kadron,bultan}@cs.ucsb.edu

†Harvey Mudd College, Claremont, CA 91711, bang@cs.hmc.edu

ABSTRACT

Information leakage is a significant problem in modern software systems. Information leaks due to side channels are especially hard to detect and analyze. In recent years, techniques have been developed for automated synthesis of adaptive side-channel attacks that recover secret values by iteratively generating inputs to reveal partial information about the secret based on the side-channel observations. Prominent approaches of attack synthesis use symbolic execution, model counting, and meta-heuristics to maximize information gain. These approaches could benefit by reusing results from prior steps in each step. In this paper, we present an incremental approach to attack synthesis that reuses model counting results from prior iterations in each attack step to improve efficiency. Experimental evaluation demonstrates that our approach drastically improves performance, reducing the attack synthesis time by an order of magnitude.

1. INTRODUCTION

Computation occurring over sensitive data in software systems can have measurable non-functional characteristics that can reveal information. This can allow a malicious user to infer information about sensitive data by measuring characteristics such as execution time, memory usage, or network delay. This type of unintended leakage of information about sensitive data due to non-functional behavior of a program is called a side-channel vulnerability. Recently developed techniques [5, 10, 14] focus on automatically synthesizing adaptive side-channel attacks against functions that manipulate secret values by generating a sequence of public inputs that a malicious user can use to leak information about a secret when observing side-channel behavior. These techniques use symbolic execution to extract constraints that characterize the relationship between the secret values in the program, attacker controlled inputs, and side-channel observations. Several methods have been investigated and compared in [14] for selecting the next attack input based on meta heuristics for maximizing the amount of information gained and automata-based techniques for constraint solving and model counting. In this paper, we present incremental attack synthesis approach, re-using the results from prior iterations in order to improve the performance of each attack synthesis step. Each and every step of adaptive attack synthesis need to count number of solutions satisfying the constraints generated on secret value, known as model counting, responsible for expensive computation of attack synthesis. Our insight is that the constraints generated in an attack step is incremental in nature. If we can reuse model counting results from earlier steps, performance of attack synthesis techniques could improve. Our contributions in this paper can be summarized as follows: (1) we present an incremental attack synthesis approach based on incremental automata-based model counting that reuses the results from prior attack steps in order to improve the efficiency of attack synthesis; (2) we can automatically synthesize side-channel attacks for programs that manipulate both numeric values and unbounded string; and (3) we present experiments on Java functions and case studies demonstrating realistic attack scenarios, and our experiments demonstrate that our attack synthesis approach is effective and our incremental approach drastically improves the performance of attack synthesis, reducing the attack synthesis time by an order of magnitude.

2. MOTIVATION

Consider a simple example of lexicographical inequality checking of two strings (Figure 1). If secret value h is lexicographically smaller than user input l, the execution time of *stringInequality* corresponds to 47 java bytecode instructions, and 62 otherwise. Symbolically executing the *stringInequality* method (note that, we do not symbolically execute the *compareTo* method from Java's string library but capture it as a string constraint directly), two path constraints are inferred with distinguishable observations shown in Table 1. For simplicity, consider the secret domain to be from "AA" to "ZZ" ($26^2 = 676$ strings), the secret value is "LL" and the first attack input is "AA". In Table 3 we show an attack that recovers the secret in 20 attack steps. We can generate an attack like the one shown in Table 3 by finding a satisfying solution (i.e., model) to the constraints on the low variable that is consistent with the observations about the secret we have accumulated so far. We call this the Model-based (M) approach (see section 3.4), which can generate optimal segment attacks [4, 9, 13] in which an attacker can learn a secret segment by segment. However, for the example in Figure 1 the Model-based approach cannot generate an optimal attack. The attack in Table 3 recovers the secret but is not optimal in terms of the length of the attack. To generate an optimal attack we must choose an input that maximizes the information gain in each step. Then, we can generate the attack shown in Table 2 which is optimal and requires only 9 steps. This corresponds to a binary search, finding the middle point to divide the domain of secret value in a balanced way. For our example, the domain size d is 26^2 and taking $\log_2 d$, we get $\lceil 9.40 \rceil = 10$ attack steps in the worst case. In order to generate the optimal attack automatically, we construct an objective function (see section 3.2) characterizing the information gain for each attack step and use optimization techniques (see section 3.4) to maximize the objective function. Now, let us have a look at the constraints on secret value h at each attack step for the optimal attack from Table 4. At each attack step we gain new information about the secret value h and a new constraint is added to the existing constraint C_h. The constraint C_h grows and becomes more complex in each attack step. Constraint solving and model counting are the most expensive parts of our approach. So, if we can reuse prior solutions to constraint solving and model counting to take advantage of the incremental nature of attack synthesis, we can increase the

```
public void stringInequality(String h, String l) {
    if(h.compareTo(l) <= 0) {
      for (int i = 1; i > 0 ; i--);
    } else {
      for (int i = 5; i > 0 ; i--);
    }
}
```

Figure 1: String inequality example.

efficiency of our approach. We call this approach *incremental attack synthesis* (see section 3.1) and demonstrate that it improves the efficiency of attack synthesis significantly (see section 4).

Table 1: Observation constraints for Figure 1.

i	ψ_i	o
1	$h \leq l$	42
2	$h > l$	67

Table 2: Optimal attack

Step	$\mathcal{H}$	l	o
1	8.40	"MZ"	42
2	7.40	"GM"	67
3	6.40	"JS"	67
4	5.43	"LI"	67
5	4.39	"MD"	42
6	3.32	"LS"	67
7	2.32	"LN"	67
8	1.00	"LK"	67
9	0.00	"LL"	42

Table 3: Non-optimal attack

Step	$\mathcal{H}$	l	o	Step	$\mathcal{H}$	l	o
1	9.40	"AC"	67	11	7.56	"PJ"	42
2	9.39	"AE"	67	12	6.82	"PI"	42
3	9.39	"JZ"	67	13	6.80	"NA"	42
4	8.70	"XE"	42	14	5.70	"LZ"	42
5	8.41	"XB"	42	15	4.64	"LI"	67
6	8.40	"KQ"	67	16	4.00	"LR"	42
7	8.33	"XA"	42	17	3.00	"LK"	67
8	8.32	"KU"	67	18	2.58	"LO"	42
9	8.30	"SI"	42	19	1.58	"LM"	42
10	7.60	"KZ"	67	20	0.00	"LL"	42

Table 4: Incremental nature of constraints at each attack step.

Attack step	Attack input	Constraint on secret value, C_h
1	"MZ"	$h <=$ "MZ"
2	"GM"	$h <=$ "MZ" $\wedge\ h >$ "GM"
3	"JS"	$h <=$ "MZ" $\wedge\ h >$ "GM" $\wedge\ h >$ "JS"
...	...	...
9	"LL"	$h <=$ "MZ" $\wedge\ h >$ "GM" $\wedge\ h >$ "JS" $\wedge$ $h >$ "LI" $\wedge\ h <=$ "MD" $\wedge\ h <=$ "LS" $\wedge$ $h >$ "LN" $\wedge\ h >$ "LK" $\wedge\ h <=$ "LL"

3. INCREMENTAL ATTACK SYNTHESIS

In this section, we first present an overview of attack synthesis. Next, we describe the objective function that guides the synthesis of each attack step. Then, we discuss automata-based model counting for computing the objective function. Finally, we describe our incremental approach to attack synthesis that reuses results of prior model counting queries to improve efficiency.

3.1 Attack Synthesis

We use a two-phase attack synthesis approach as shown in Fig. 2. We consider a function F that takes as input a secret $h \in \mathbb{H}$ and an attacker-controlled input $l \in \mathbb{L}$ and that generates side-channel observations $o \in \mathbb{O}$. We perform symbolic execution on F with the secret (h) and the attacker controlled input (l) marked as symbolic [12]. During symbolic execution, we keep track of a side-channel observation for each path. For example, for timing side-channels, we model the execution time of the function by the number of instructions executed like existing works [4,10,11]. We assume that the observable values are noiseless. We augmented symbolic execution to return a function that maps a path constraint ϕ to an observation o. We combine observationally similar path constraints via disjunction, where we say that o and o' are in the same equivalence class ($o \sim o'$) if and only if $|o - o'| < \delta$. The resulting *observation constraints* (denoted ψ_o and Ψ) characterize the relationship between the secret (h) the attacker input (l) and indistinguishable side-channel observations (o).

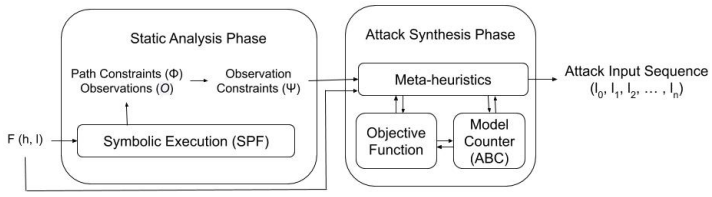

Figure 2: Overview of Attack Synthesis Approach

To synthesize attack, 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. For each attack step, we determine the input value l^*. Then, the observation o that corresponds to running the program under attack with h^* and l^* is revealed by running the function using the public input l^*. 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. Based on C_h, we compute an uncertainty measure for h at every step using Shannon entropy, denoted $\mathcal{H}$ (Section 3.2). The goal is to generate inputs which drive $\mathcal{H}$ as close as possible to zero. This attack synthesis phase is repeated until it is not possible to reduce the uncertainty, $\mathcal{H}$, any further.

3.2 Objective Function for Information Gain

Here we provide an objective function to measure the amount of information an attacker expects to gain for an input value l_{val}. Let H, L, and O be 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 [15]. Given probability function $p(h)$, the *information entropy* of H, denoted $\mathcal{H}(H)$, which we interpret as the initial *uncertainty* about the secret, 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 choice of $L = l_{val}$, with expectation taken over all observations, $o \in O$. We compute the *conditional entropy of H given O and l_{val}* as

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

Then, we use Equation 3 as our objective function which computes the expected amount of information *gained* about h by observing o after running the function F with a specific input 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 $L = l_{val}$.

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

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 $p(h|o,l)$ 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, which gives the number of satisfying solutions for a formula f, denoted $\#f$.

Thus, we observe that $p(h) = 1/\#C_h$ if h satisfies C_h and is 0 otherwise. Hence, Equation 1 reduces to $\mathcal{H}(H) = \log_2(\#C_h)$. Our augmented symbolic execution gives us 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 the secret has a particular 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, $p(h|o_i, l_{val}) = 1/\#(C_h \wedge \psi_i)[l \mapsto l_{val}]$. Similarly, $p(o_i|l_{val}) = \#(C_h \wedge \psi_i)[l \mapsto l_{val}]/\#C_h[l \mapsto l_{val}]$. In this paper, $p(h)$, $p(o|l)$, and $p(h|o, l)$ are computed using the ModelCount algorithm described in the next section. An attacker can optimize the information gain by trying many different l_{val} values and computing the corresponding MutualInfo.

3.3 Automata-Based Model Counting

We use and extend the Automata-Based Model Counter (ABC) tool, which is a constraint solver for string and numeric constraints with model counting capabilities [2]. The constraint language for ABC supports all numeric constraints solved by off the shelf constraints solvers as well as typical string operations such as *charAt, length, indexOf, substring, begins, concat,* $<$, $=$, etc. Given a constraint C, ABC constructs a multi-track deterministic finite automaton (DFA) A_C that characterizes all solutions for the constraint C, where $\mathcal{L}(A_C)$ corresponds to the set of solutions for C. For each string term γ or integer term β in the constraint grammar [3], ABC implements an automata constructor function which generates an automaton A that encodes the set of satisfying solutions for the term. ABC implements specialized DFA construction algorithms for atomic string operations. ABC counts the number of models for a constraint C by first constructing the corresponding automaton A_C and using the observation that number of strings of length k in $\mathcal{L}(A_C)$ is equal to the number of accepting paths of length k in the DFA A_C. Consequently, ABC treats the DFA A_C that results from solving C as a graph where DFA states are graph vertices and the weight of an edge is the number of symbols that have a transition between the source and destination vertices (states) of that edge. A dynamic programming algorithm that computes the kth power of the adjacency matrix of the graph is used to count the number of accepting paths in the DFA of length k (or less than or equal to k) [2].

3.4 Incremental Model Counting

Attack synthesis requires solving and model counting the constraint on the secret, C_h, and updating it with the current instantiated observation constraint, $\psi_o[l \mapsto l_{val}]$, resulting in a new constraint, $C_h \wedge \psi_o[l \mapsto l_{val}]$, for which then compute the entropy. As this process is executed many times, multiple calls to ABC occur, often with similar constraints. In each iteration, ABC is re-solving each sub-constraint from scratch, constructing a DFA for each of them, then combining them using DFA intersection. Note that, during attack synthesis, C_h can become a complex combination of constraints that represent what we learned over the course of the attack. Then ABC would be unnecessarily re-solving the subconstraints of C_h in each attack step. To summarize, we observe that, during attack synthesis: 1) the constraint that characterizes the set of secrets that are consistent with the observations and low inputs (C_h) is constructed incrementally, and 2) computing entropy using incremental constraint solving can improve the performance by exploiting the incremental nature of attack synthesis. We implemented incremental constraint solving and model counting by extending ABC so that it retains state: given a constraint, C, ABC constructs the automaton representing the set of solutions to C, which is then stored for use in later calls. The steps of attack synthesis involve two types of model counting for C_h and $\psi_o[l \mapsto l_{val}]$: during MutualInfo when an attacker optimizes the attack by trying many different l_{val}, and in Entropy, during computation of the remaining uncertainty. In both situations, model counting is required on many different constraints, and most of the sub-constraints come from previous iterations. We augmented ABC with an interface so that, given $C_h \wedge \psi_o[l \mapsto l_{val}]$, we can check if an automaton has already been constructed for either C_h or $\psi_o[l \mapsto l_{val}]$, and if so, to get the already constructed automata for them, rather than re-solving each constraint. Note that for the purposes of model counting, $\psi_o[l \mapsto l_{val}]$ can be represented as $\psi_o \wedge l = l_{val}$. Our incremental model counting approach is outlined in Algorithm 1. Given the constraint $C_h \wedge \psi_o \wedge l = l_{val}$, GetDFA retrieves the previously constructed automaton for C_h, A_{C_h}. Algorithm 1 is called with a new observation constraint ψ_o in each attack step, for which the automaton must first be constructed. Subsequent calls with the same ψ_o use the previously constructed automaton. A new $A_{l=l_{val}}$ must be constructed for each model counting query (as each query involves a different l_{val}). The final automaton A is constructed using automata product from $A_{C_h}, A_{\psi_o}, A_{l=l_{val}}$. A is exactly the same automaton constructed from $C_h \wedge \psi_o \wedge l = l_{val}$, but it is constructed incrementally, thus allowing re-use of previously constructed automata.

Algorithm 1 ModelCountIncremental($C_h \wedge \psi_o \wedge l = l_{val}$)
Incremental model counting for constraint $C_h \wedge \psi_o \wedge l = l_{val}$.

1: $A_{C_h} \leftarrow$ GetDFA(C_h)
2: **if** IsConstructed(ψ_o) **then** $A_{\psi_o} \leftarrow$ GetDFA(ψ_o)
3: **else** $A_{\psi_o} \leftarrow$ Construct(ψ_o)
4: $A_{l=l_{val}} \leftarrow$ Construct($l = l_{val}$)
5: $A \leftarrow A_{C_h} \cap A_{\psi_o} \cap A_{l=l_{val}}$
6: **return** ModelCount(A)

3.5 Attack Synthesis Heuristics

At every attack step the attacker's goal is to choose a low input l^* that reveals information about h^*. Meta heuristic approaches explore a subset of the possible low inputs. Constraint solving and meta heuristic techniques are used to synthesize attack inputs l^*. Different techniques have been investigated in our earlier work [14]. The simplest approach of attack synthesis is to generate a single random model and use it as the next attack input know as Model-based (M) approach. In this paper, we consider only Simulated annealing (SA) as a meta-heuristic for optimizing an objective function.

4. IMPLEMENTATIONS AND EXPERIMENTS

Implementation. Our implementation consists of two components, corresponding to the two phases described in section 3. We use Symbolic Path Finder (SPF) [12] for symbolic execution and implemented out attack synthesis algorithm as a Java program that takes the generated observation constraints as input, along with C_h, h^*. We implemented ModelCount, and ModelCountIncremental by extending the existing string model counting tool ABC as described in section 3.1. 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 9 Java canonical programs (Table 5). Functions `PCI` and `PCS` are password checking implementations. Both compare a user input and secret password but early termination optimization (as described in section 1) induces a timing side channel for the first one and the latter is a constant-time implementation. We analyzed the `SE` and `IO` methods from the Java String library which is known to

Table 5: 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	16	1
stringEquals	SE	charAt,length	4	4	9	9
stringInequality	SI	$<,\geq$	4	4	2	2
stringConcatInequality	SCOI	concat,$<,\geq$	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

contain a timing side channel. Function ED is an implementation to calculate edit distance of two strings. Function CO is a compression algorithm which collapses repeated substrings within two strings. SI, SCOI and SCI check lexicographic inequality ($<, \geq$) of two strings in different ways.

Experiment Setup. For experiments, we used a machine with an Intel Core i5-2400S 2.50 GHz CPU, 32 GB of DDR3 RAM, running Ubuntu 16.04. We used the OpenJDK 64-bit Java VM. We ran each experiment for 5 random secrets, with mean values of the results in Table 6. For SA, the temperature range (t to t_{min}) is from 10 to 0.001 and cooling rate k is 0.1.

Results. In this discussion, we describe the quality of a synthesized attack according to these metrics: attack synthesis time, attack length, and overall change in uncertainty about the secret measured as entropy from $\mathcal{H}_{init}$ to $\mathcal{H}_{final}$ and efficiency of incremental attack synthesis in terms of time. Attacks that do not reduce the final entropy to zero are called *incomplete*. Incomplete attacks are mainly due to one of two reasons: the program is not vulnerable to side-channels (for example PCS) or the observation constraints are very complex, combining lots of path constraints which slows progress too much so that not enough information is leaked within the given time bound (for example ED and SCI). For the purpose of direct comparison, in our experiments, we set a bound of 5 hours for SA (slowest technique) on ED and SCI and computed a bound for $\mathcal{H}_{final}$ of **17.28** and **14.48**, respectively, while all other examples reduced $\mathcal{H}_{final}$ to **0.0**. These examples are marked with $*$. Note that, M and SA-I can reduce $\mathcal{H}_{final}$ for ED and SCI to **14.34** and **12.28**, respectively, after one hour.

Attack Synthesis Time Comparison. We observe that the model-based technique (M), which only uses C_l to restrict the search space is faster than other techniques, as it greedily uses a 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 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. Figure 3 shows how M can synthesize attacks faster compared to SA (in seconds).

Attack Length Comparison. Although M is fast in synthesizing attacks for each benchmark, experimental results show that it requires more attack steps (in terms of information gain) than meta-heuristic techniques that optimize the objective function. As the experimental results show for the SI, SCOI and SCI, 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. If attacker is aware of the "flat" objective function phenomenon, they can proceed with M. In general, M is not efficient to generate an attack with reduced number of attack steps and hence, meta heuristics like SA approach are required. Figure 4 shows how SA is better than M in terms of length of the generated attacks. Note that, we say M vs SA/SA-I as incremental version will make difference in attack synthesis time, not attack length.

Efficiency of Incremental Attack Synthesis. On one hand, we can synthesize attacks faster using M but attacks synthesized by M require more attack steps in general. On the other hand, we can synthesize attacks with minimal number of attack steps using SA, but attack synthesis process is slower for SA. Our experiments demonstrate that incremental SA gives us fast attack synthesis without increasing the attack length. We compare incremental SA (SA-I) against SA. Figure 5 shows SA-I is an order of magnitude faster than SA for all the examples from the benchmark. We also compare SA-I against M and Figure 6 shows that SA-I is comparable to M in terms of attack synthesis time (in seconds).

Vulnerability to Side-Channels. Finally, we observe that some of our benchmarks are more secure against our attack synthesizer than others. In particular, PCS, a constant-time implementation of password checking, did not leak any information through the side channel. One of the examples from the benchmark, ED also did not succumb to our approach easily, due to the relatively large number of generated constraints (2170), 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, providing evidence of vulnerability (e.g. PCI). Further, when attack synthesizer fails to generate attacks (PCS), or is only able to extract a relatively small information after many steps of significant computation time (ED), it provides evidence that the function under test is comparatively safer against side-channel attacks. Table 6 shows results for PCS, hardly reducing $\mathcal{H}_{final}$ even after running for 1 hour for M, SA and SA-I.

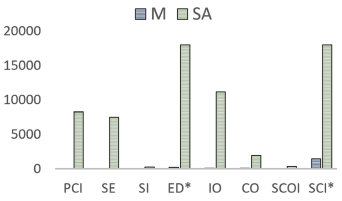

Figure 3: Attack Synthesis Time, M vs SA

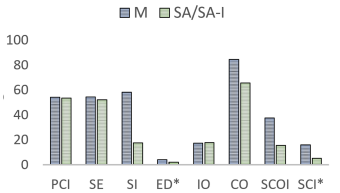

Figure 4: Attack Length, M vs SA/SA-I

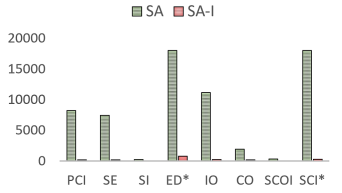

Figure 5: Attack Synthesis Time, SA vs SA-I

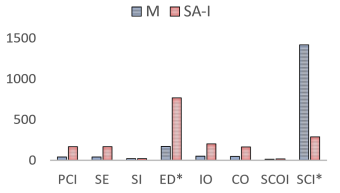

Figure 6: Attack Synthesis Time, M vs SA-I

Table 6: Secure password checker (PCS) results, 3600s time bound.

ID	$\mathcal{H}_{init}$	Metrics	M	SA	SA-I
PCS	18.8	Steps	108	14	99
		$\mathcal{H}_{final}$	18.8	18.8	18.8

5. CASE STUDIES

CRIME Attack. The "Compression Ratio Info-leak Made Easy" (CRIME) attack [13] allows an attacker to learn fields of encrypted web session headers by injecting extraneous text (l) into a procedure that compresses and encrypts the header (h). Despite the encryption, an attacker can infer how much of the injected text matched the unknown header by observing the number of bytes in the compressed result [4,13]. Our approach automatically synthesizes this attack. Symbolic execution of the compression function (LZ77T) for a secret of length 3 and alphabet size 4 yields

187 path constraints and 4 observations, leading to 4 observation constraints. M synthesizes an attack in 6.8 steps within 468.5 seconds. SA-I could generate the attack in 7.8 steps within 757.4 seconds. SA-I does not improve over M due to "flat" objective function. Note that [4] performs leakage quantification for this example but does not synthesize attacks.

Law Enforcement Database. The Law Enforcement Employment Database Server is a network application included as a part of the DARPA STAC program [1]. This application provides access to law enforcement personnel records.The database contains restricted and unrestricted employee information. Users can search ranges of employee IDs as database keys. If an ID query range contains one or more restricted IDs, the returned data will not contain the restricted IDs. We decompiled the application and then symbolically executed the `channelRead0` method from the `UDPServerHandler` class which performs the search operation. We limited the domain of ids to 1024, added 30 unrestricted IDs and 1 restricted ID. Symbolic execution gives 1669 path constraints with 162 distinguishable observations ($\delta = 10$ instructions). M generates attack with an attack length of 8.2 in 270.1s whereas SA-I generates an attack with length of 6.5 in 810.7s. SA-I requires less steps as the objective function is not "flat".

6. RELATED WORK

Recent works address either synthesizing attacks or quantifying information leakage under a model where the attacker can make multiple invocations of the system [4,5,7,11]. There has been work on multi-run analysis using enumerative techniques [7]. None of these earlier results present a symbolic and incremental approach to adaptive attack synthesis as we present in this paper. Among other existing works, [4] assumes that the given program has a segment oracle side-channel vulnerability. [10] works only for linear arithmetic and bit-vector constraints. [14] focuses on studying different meta-heuristic techniques such as random search, genetic algorithm, and simulated annealing for string functions. In contrast, our approach is more general and can handle any program with numeric, string, and mixed constraints. Existing approaches use constraint solving and model counting queries to quantify the information leakage, but they do not use an incremental approach and, therefore, re-compute many sub-queries. Due to the importance of model counting in quantitative program analyses, model counting constraint solvers are gaining increasing attention [2,6,8]. Among these works, [6] focuses on subformula caching generalizing quantitative program analysis problems whereas our approach exploits incremental nature of attack synthesis, being more efficient. ABC [2] is the only one that supports string, numeric, and mixed constraints. We extended ABC to perform incremental model counting for our attack synthesis approach. Other work in quantifying information leakage [5, 10, 11] have used symbolic execution and model-counting techniques for linear integer arithmetic.

7. CONCLUSION

In this paper, we exploit the iterative nature of attack synthesis by presenting an incremental approach that reuses results from prior iterations. We implemented our attack synthesis approach for Java programs using symbolic execution tool SPF and ABC model counter. We evaluated the effectiveness of our attack synthesis approach on several functions and two case studies. Our experiments demonstrate that our incremental approach reduces attack synthesis time in order of magnitude.

8. REFERENCES

[1] `https://github.com/Apogee-Research/STAC/tree/master/Engagement_Challenges`.

[2] 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.

[3] Abdulbaki Aydin, William Eiers, Lucas Bang, Tegan Brennan, Miroslav Gavrilov, Tevfik Bultan, and Fang Yu. Parameterized model counting for string and numeric constraints. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 400–410. ACM, 2018.

[4] 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.

[5] 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.

[6] William Eiers, Seemanta Saha, Tegan Brennan, and Tevfik Bultan. Subformula caching for model counting and quantitative program analysis. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE) (to appear)*, 2019.

[7] 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.

[8] Loi Luu, Shweta Shinde, Prateek Saxena, and Brian Demsky. A model counter for constraints over unbounded strings. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 57, 2014.

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

[10] 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.

[11] 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.

[12] 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.

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

[14] Seemanta Saha, Ismet Burak Kadron, William Eiers, Lucas Bang, and Tevfik Bultan. Attack synthesis for strings using meta-heuristics. *SIGSOFT Softw. Eng. Notes*, 43(4):56–56, January.

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