2016 IEEE 29th Computer Security Foundations Symposium

Multi-run side-channel analysis using Symbolic Execution and Max-SMT

Corina S. Păsăreanu
Carnegie Mellon University, NASA Ames
Moffet Field, CA, USA
corina.s.pasareanu@nasa.gov

Quoc-Sang Phan
Carnegie Mellon University
Moffet Field, CA, USA
sang.phan@sv.cmu.edu

Pasquale Malacaria
Queen Mary University of London
London, United Kingdom
p.malacaria@qmul.ac.uk

***Abstract*—Side-channel attacks recover confidential information from non-functional characteristics of computations, such as time or memory consumption. We describe a program analysis that uses symbolic execution to quantify the information that is leaked to an attacker who makes multiple side-channel measurements. The analysis also synthesizes the concrete public inputs (the "attack") that lead to maximum leakage, via a novel reduction to Max-SMT solving over the constraints collected with symbolic execution. Furthermore model counting and information-theoretic metrics are used to compute an attacker's remaining uncertainty about a secret after a certain number of side-channel measurements are made. We have implemented the analysis in the Symbolic PathFinder tool and applied it in the context of password checking and cryptographic functions, showing how to obtain tight bounds on information leakage under a small number of attack steps.**

***Index Terms*—Side-Channel Attacks; Quantitative Information Flow; Cryptography; Multi-run Security; Symbolic Execution; Satisfiability Modulo Theories; Max-SMT**

I. Introduction

Side-channel attacks recover secret inputs to programs from non-functional characteristics of computations, such as time consumed, number of memory accesses or size of output files. Side-channel attacks have been shown to pose serious threats by recovering cryptographic keys, e.g. when using the well known RSA encryption/decryption algorithm [1], and private information about users, e.g. as with commonly used algorithms for data compression [2].

We propose a symbolic execution approach for the automatic analysis of software that computes *quantitative* bounds on the amount of information that can be leaked via side-channel attacks. Technically we use the fact that the amount of leaked information corresponds to the number of different possible side-channel observations, which we compute using symbolic execution and model counting via a reduction to symbolic quantitative information flow analysis (QIF)[3], [4]. The analysis is parametrized by a *cost model* which allows us to obtain side-channel measurements (time, memory, bytes written to a file, etc.) from the execution of bytecode instructions. The "observables" are the values for the cost computed for each path in the analyzed program. Furthermore we provide a method for automatically deriving the public user input that results into *maximum leakage* by using weighted Max-SMT solving for which efficient procedures exist [5].

Our key insight is that Max-SMT solving can be used to obtain the maximal assignment over the set of clauses obtained with symbolic execution (i.e. any other assignment satisfies less clauses) and this corresponds to the largest number of observables that can be reached by a particular public input, hence is the maximal leakage: any other choice of public input would result in less observables. We show experimentally that this new Max-SMT encoding can be more efficient than established approaches based on bounded model checking over program self-compositions [6] or on enumerating over the concrete values.

Our Max-SMT approach generalizes naturally over multiple-run side-channel attacks where we use symbolic execution to quantify the information revealed to an attacker after multiple channel measurements made. This corresponds to a typical scenario where an attacker makes multiple guesses by invoking and measuring the execution of the program multiple times on different public inputs to gradually uncover a secret that is constant across program runs (such as the secret key used in the RSA encryption/decryption algorithm). Max-SMT solving is used to compute a *sequence* of public inputs that lead to maximum leakage, exposing the vulnerability of the program to multi-run attacks.

Furthermore we show how to use an extension of symbolic execution, namely with *quantitative reasoning* [7], [8], to compute precise values for information theoretic metrics such as Shannon or Smith's min entropy [9]. The technique uses model counting over the constraints collected by symbolic execution, to compute the probability of executing different program paths (under a user-specified profile). Thus we can compute an attacker's remaining uncertainty about a secret after a number (k) of side-channel measurements made. We can also determine whether a secret is fully revealed after k runs or whether a program keeps leaking information after k runs etc.

We have implemented the analysis in the Symbolic PathFinder tool and show how to use it to measure side-channel vulnerabilities for Java bytecode programs in the presence of non-adaptive attacks. Our approach is general and can be implemented easily in other symbolic execution tools targeting other languages. We discuss the application of our approach in the context of password checking and

DOI 10.1109/CSF.2016.34

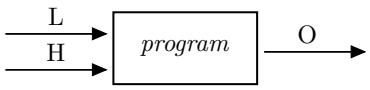

Fig. 1. Attacker Model

cryptographic functions, showing how to obtain tight bounds on information leakage under a small number of attack steps.

The rest of the paper is organized as follows. In the next section we give background on quantitative information flow analysis, symbolic execution and constraint solving. We next discuss how to perform side-channel analysis using symbolic execution. Section IV outlines our Max-SMT encoding for finding the low input that maximizes leakage. Section V describes extending the analysis over multiple runs and Section VI discusses treatment of multi-threading. Section VII describes our implementation and experiments, Section VIII gives closely related work and Section IX gives our conclusions.

II. Background

A. Quantitative Information Flow Analysis

Quantitative Information Flow (QIF) is a powerful approach to *"measure"* leaks of confidential information in a software system. Typically simple, qualitative, information flow analysis accepts programs as secure if confidential data cannot be inferred by an attacker through their observations of the system—this intuitive property is called *non-interference*. Although satisfying non-interference is a sound guarantee for a system to be secure, this requirement is too strict for most realistic programs: to leak some information is almost unavoidable. By quantifying leakage QIF addresses this limitation: not only programs with "zero interference" (non-interference) can be accepted as secure, but also programs with "small" interference.

Fig. 1 shows an example function that we use to illustrate QIF. It is a convention in the security literature to use the label L (low) to denote non-sensitive input, to use the label H (high) to denote sensitive private input, and to use the label O (output) to denote the output. A malicious user has access to the public data, L and O, and tries to infer the hidden secret, H, from that.

Similar to previous work[10] we assume that the malicious user can make one side-channel measurement per invocation of the program P and that no measurement errors occur. Furthermore, we assume that the attacker has full knowledge about the implementation of P.

A fundamental QIF result (the channel capacity theorem [11], [9]) shows that leakage for a program is always less than or equal to the log of the number of possible distinct observations that an attacker can make. By noting these observables with ($NObs$) and channel capacity with CC, we have hence:

$$\text{Information leaked} \leq \log_2(NObs) = CC(P)$$

The result states in essence that QIF reduces to *counting* the number of different observable outputs for the program. The result holds for different notions of leakage based on the probability of guessing the secret or the notion of leakage based on Shannon's information theory measuring the number of bits leaked. For these reasons counting the number of observables is the basis of state-of-the-art QIF analysis [6], [12], [13], [14], [15], [16].

B. Symbolic Execution

Symbolic execution is a program analysis technique which executes programs on symbolic rather than concrete inputs, and it computes the program effects as *functions* in terms of the symbolic inputs [17]. The behavior of a program P is determined by the values of its inputs and can be described by means of a *symbolic execution tree* where tree nodes are program states and tree edges are the program transitions as determined by the symbolic execution of program instructions.

The state s of a program is defined by the tuple (IP, V, PC) where *IP* represents the next instruction to be executed, V is a mapping from each program variable v to its symbolic value (i.e., a symbolic expression in terms of the symbolic inputs), and *PC* is a *path condition*. *PC* is a conjunction of constraints over the symbolic inputs that characterizes those inputs that follow the path from the program's initial state to state s. The current state s and the next instruction *IP* define the set of transitions from s. The feasibility of the path conditions is checked using off-the-shelf constraint solvers [18], as the conditions are encountered during symbolic execution to detect infeasible paths and to generate test inputs that execute feasible paths. There are many tools that support symbolic execution for programs [19], [20], [21]. For our implementation we use Symbolic PathFinder – part of Java PathFinder tool set – described below.

C. Java PathFinder and Symbolic PathFinder

Java PathFinder [22] (JPF) is an extensible run-time environment for the verification of Java bytecode, i.e., compiled Java programs. The analyses proposed here are incorporated Symbolic PathFinder (SPF) [23], which extends JPF with a symbolic execution mode. SPF uses JPF to systematically generate and execute the symbolic execution tree of the code under analysis and also to handle multi-threading. SPF uses a variety of off-the-shelf decision procedures and constraint solvers to solve the constraints generated by the symbolic execution of bytecode programs.

In recent work [7], [8] SPF was extended with *quantitative reasoning* which we leverage for quantifying the results. Specifically, SPF uses a combination of symbolic execution and model counting [24], [8] for computing the *probability* of successful termination (and alternatively the probability of failure) for a Java software component placed in a stochastic environment.

D. SMT and weighted Max-SMT

A propositional variable p is an assertion that must be true or false. A propositional variable p or its negation $\neg p$

```
int c = 32, low = 0, time = 0;
 while (c >= 0) {
 int m = (int)Math.pow(2,c);
 if (high >= m) {
  low = low + m;
  high = high - m;
  time++;
 }
 c = c - 1;
 time++;
}
low = 0;
```

Fig. 2. Example of timing channel

is called a *literal*. A *clause* is a disjunction of literals. A *propositional formula* is a conjunction of clauses. The problem of *propositional satisfiability* (SAT) is to determine, for a given formula, whether or not it is satisfiable, i.e. if it has a *model*: an assignment of Boolean values to variables that satisfies the formula.

An extension of SAT is the *satisfiability modulo theories* (SMT) problem [25]: which decides the satisfiability of a given quantifier-free first-order formula with respect to a background theory, such as linear arithmetic.

Max-SAT [26] is an extension of SAT to optimization. Given a *weighted* formula φ_w where each clause C_i has a weight ω_i (positive or infinity), find the assignment such that the sum of the weights of the falsified clauses is minimized.

The Max-SMT problem [5] is a generalization of the Max-SAT problem to first-order theories: given a weighted first-order formula φ_w, find a model that minimizes the sum of the weights of the falsified clauses, or alternatively maximizes the sum of satisfied clauses; we use the latter formulation here. Several powerful SMT and Max-SMT solvers exist [18], [5] and have been shown to work on industrial strength examples.

III. Symbolic Execution for Side-Channel Analysis

Given a program P with "high" inputs (secret) and "low" inputs (public) our goal is to compute information leakage via side-channels, i.e. time or memory consumption, writing to a file or socket etc. We base the computation of the leakage on Shannon's Information theory where the *observables* $\mathcal{O}$ are the *costs* computed for each program path π, denoted $cost(\pi)$ i.e. number of instructions executed along π, number of bytes allocated etc.

Thus we assume that each path can only give one observable. Our work is done in the context of a project that specifically targets side-channels but it is applicable to more general quantitative information flow analysis where this assumption holds.

This approach is illustrated by the example in Fig. 2. This program would leak all secrets (denoted by "high") into "low", but since "low" is set to 0 at the end of the program, the leakage into "low" is eliminated. However there is a side time channel which reveals the number of 1s in the binary representation of the variable "high" (the side channels used to break RSA had this shape [27]). That side channel can be captured by introducing an auxiliary "time" variable as in Fig. 2. Intuitively, by introducing the variable "time" we simulate an adversary observing the timing channel. By observing the variable "time" at the end of the program we can deduce the timing leakage of "high" so to recover the side channel in terms of classical observables [3], [4]. Similarly, for memory allocation we keep a variable that counts the number of bytes allocated for example, according to the *cost model* yielding a unified framework for detecting time/memory side channels and for quantifying how many bits of information are being leaked.

Notice that in practice we do not introduce an auxiliary variable for time (or memory) but measure directly the cost of executing each bytecode instruction, using JPF listeners (see Section VII).

For the example in the Fig. 2, our analysis gives the result that the leakage measured in channel capacity is 5.459 bits.

A. Computing Channel Capacity

To compute the Channel Capacity we perform a symbolic execution over the program where both "high" h and "low" l inputs are symbolic. We assume that the analyzed program is deterministic and that the input domain is finite.

We run symbolic execution to collect all symbolic paths of the program. Note that there is only a finite number of them. Each symbolic path *summarizes* multiple (possibly many) concrete program paths that follow the same instructions through the analyzed code. For simplicity we will assume that all the paths terminate within the prescribed bound. As this is not always the case in general, in practice we can use a notion of *confidence* (similar to [7]) that quantifies the impact of the execution bound on the quality of the analysis.

Let $P(h,l)$ be the (deterministic) program, and $\pi_1, \pi_2..\pi_n$ be the symbolic paths of $P(h,l)$. Let $PC_1(h,l)$, $PC_2(h,l)\ldots$ $PC_n(h,l)$ be the path conditions (disjoint by construction) computed with symbolic execution. Throughout the paper we use lower-case h and l to denote symbolic high, respectively low, inputs. Upper-case H and L denote concrete values. Assume every PC contains at least one high variable (if not we eliminate all the PC's that only depend on low variables since they leak no information). Note that h and l may represent vectors of secrets and public values, respectively.

We can then extract the number of observables by *counting* the number of paths that have different cost. For each symbolic path π_i we compute its cost $cost(\pi_i)$ representing the side-channel measurement along that path according to a cost model.

Let $\mathcal{O} = \{o_1, o_2, ...o_m\}$, with $m \leq n$, be the set of observable costs. Note that different symbolic paths may have the same cost but a path can not have more than one cost. For the rest of the paper we use cost and observable interchangeably to refer to the side-channel measurement made along a path. Since each path condition PC_i has a one-to-one correspondence to a path π_i, we also write $cost(PC_i)$ to mean $cost(\pi_i)$. Furthermore, we write $cost(P(H,L))$ to denote the cost of the (concrete) path followed by the program

on concrete values H and L. The Channel Capacity, i.e. the maximal possible leakage is then:

$$CC(P) = log_2(|\mathcal{O}|)$$

Note that this is an *upper bound* of the actual leakage of the program. We discuss how to obtain tighter bounds in the next section.

B. Computing Shannon Entropy

The Shannon entropy $\mathcal{H}(P)$ gives a precise bound for the leakage, when the secret distribution is known. We can compute $\mathcal{H}(P)$ using probabilistic symbolic execution [7] as implemented in SPF. Let $\mathcal{O} = \{o_1, o_2, ...o_m\}$ be the set of observables as defined before. For a uniform distribution over the secret, the probability of observing o_i is:

$$p(o_i) = \frac{\sum\limits_{cost(\pi_j)=o_i} \sharp(PC_j(h,l))}{\sharp D}$$

Here $\sharp(PC_j(h,l))$ denotes the number of solutions (i.e. high and low concrete values) of constraint $PC_j(h,l)$ (computed with an off-the-shelf model counting procedure such as Latte) and $\#D$ denotes the size of the input domain D assumed to be (possible very large but) finite.

The Shannon entropy is then:

$$\mathcal{H}(P) = -\sum_{i=1,m} p(o_i)\log_2(p(o_i))$$

For deterministic systems, the Shannon entropy also gives a measure of the leakage of the side-channel, corresponding also to the observation gain (on the secret) after one round of observation (see also Section V-B). Similar calculations can be used to compute other information-theoretic metrics, e.g. one can use the computed solution spaces to estimate the number of tries needed to guess a secret.

Note that SPF already incorporates Latte [24] for quantifying the solution spaces of mathematical linear-integer constraints; SPF also uses several optimizations such as simplifications, separate solving of independent constraints, caching and composition of results [8] to perform the model counting efficiently. For the example in the Fig. 2, our analysis computes the Shannon entropy: 3.925 bits (we assume the domain of variables is $0 \ldots 10^4$, uniform distribution).

For the rest of the paper we focus on the computation of leakage using the channel capacity, since it gives worst-case bounds. However our implementation supports computing Shannon entropy as well.

IV. Single-Run Analysis (Algorithm **MaxLeak**)

In the previous section we presented a symbolic execution technique for computing the side-channel leakage after one program run. However, in the analysis we did not distinguish between high and low values, resulting in an over-approximation in the computed bounds. The problem is illustrated by the example in Fig. 3.

```
void example1(int l,int h){
  if (l<0) {
    if (h<0) cost=1;
    else if (h<5) cost=2;
    else cost=3;
  } else {
     if (h>1) cost=4;
     else cost=5;
  }
}
```

Fig. 3. Example 1

```
void example2(int l,int h){
  if(h>0} {
   if(l<0) cost=1;
   else cost=2;
  }
  else {
   if(l<5) cost=1;
   else cost=4;
  }
}
```

Fig. 4. Example 2

Here `l` denotes a public input while `h` denotes the secret; `cost` captures the observed cost along each path in the program. The number of possible observables for this program is 5, yielding a leakage of $\log_2(5)$ according to the formulation from the previous section. However note that there is no public input that can possibly achieve this leakage in a single program run. Indeed, for $l < 0$ there are 3 possible observables (i.e. different costs), while for $l \geq 0$ there are 2 observables. Thus the maximum leakage of the program after one round of observations is only $\log_2(3)$.

Our goal is to compute automatically the value of the "low" public input that results in the *maximum* number of observables. This has a very important security meaning: it reveals the most vulnerable behaviors of a program in one run and the low input that trigger that behavior. Furthermore, that low input value can be used for building multiple-run attacks as described in next section.

The computation becomes more involved because for the same concrete low input, there may be multiple symbolic paths with different costs, corresponding to different values of the high input. For example, for any negative values of l, there are three symbolic paths corresponding to $h < 0$ (with cost 1), $h \geq 0 \wedge h < 5$ (with cost 2) and $h \geq 5$ (with cost 3), respectively.

Intuitively picking such a low value translates in the maximum information about a secret that an attacker can get from observing one run of the program. Let us make this intuition more precise.

A. Attackers knowledge

To find out the value of a secret H, an attacker picks a value for the public input L, invokes the program $P(H, L)$ and observes the cost obs. In general the attacker can not simply deduce H from obs but she can infer the *constraints* on the secret that are *coherent* with the observation obs.

Following [10] we say that a secret H is coherent with obs under L whenever $P(H, L)$ has cost obs. Two secrets H_1 and H_2 are *indistinguishable* under L if both $P(H_1, L)$ and $P(H_2, L)$ yield the same cost, i.e. $cost(P(H_1, L)) = cost(P(H_2, L))$. For every cost obs, the set of secret values that are coherent with obs under L forms an equivalence class of indistinguishability under L.

We propose to use symbolic execution to compute the constraints on the secret h that are coherent with observation

obs under L. Notice that all the paths that lead to same cost obs under input L satisfy the following constraint.

$$\bigvee_{cost(PC_i)=obs} PC_i(h, L)$$

Here $PC_i(h, L)$ denotes the constraint $PC_i(h, l)$ where symbolic value l was replaced with concrete value L.

It follows that, for given L, the set of observables (or equivalence classes) induced by L is given by:

$$\mathcal{O}_L = \{obs \in \mathcal{O} | \exists H. \bigvee_{cost(PC_i)=obs} PC_i(H, L)\}$$

The "most powerful attacker" will want to pick a value L that induces the partitioning with the maximum number of equivalence classes (i.e. maximum number of observables), since that will reveal the most information about the secret. Such a *maximal* L would need to satisfy the maximum number of clauses of the form $\bigvee_{cost(PC_i)=obs} PC_i(h, l)$, for $obs \in \mathcal{O}$.

One solution would be to enumerate all the possible values for the low input and for each low input L to compute the cardinality of $\mathcal{O}_L$, $|\mathcal{O}_L|$. The low input that yields the maximum number of observables is then returned to the user. Although the input domain is assumed to be finite in practice this explicit enumeration approach would be inefficient. We present a symbolic approach below.

B. Max-SMT formulation

We first rename each path condition as follows: for each $PC_i(h, l)$, we define $PC_i(h_i, l)$ where all the high symbolic values h have been renamed to fresh symbolic values h_i. Intuitively the renamed path conditions define constraints on low variables (while the high variables are left unconstrained) and the goal is to find the low input value that leads to the maximum number of observations for any value of h. This intuition has similarities with self composition [28], [29].

For example, for path condition $PC_1(h, l) : l < 0 \wedge h < 0$, we re-write it as $PC_1(h_1, l) : l < 0 \wedge h_1 < 0$ while path condition $PC_2(h, l) : l < 0 \wedge h \geq 0 \wedge h < 5$, we re-write it as $PC_2(h_2, l) : l < 0 \wedge h_2 \geq 0 \wedge h_2 < 5$ etc.

We formulate a Max-SMT problem by building a clause for each cost, where the clause is the disjunction of the PCs that lead to same cost, and the weight is 1. Intuitively each clause defines an indistinguishability equivalence relation as defined above. We then let Max-SMT solver pick the value L that satisfies most clauses, meaning that it induces the partitioning on the secret with the maximum number of equivalence indistinguishability classes. Algorithm 1 outlines our approach, where the input is a program $P(h, l)$ and a function $cost$ that defines the cost for each program path.

Theorem 1: The solution returned by Algorithm 1 yields maximum leakage for program P.

Proof 1: Let L be the solution for l, w be the total weight and $C_{i_1}, C_{i_2}..C_{i_w}$ be the set of satisfiable clauses returned by Max-SMT. L induces a partitioning where each equivalence class contains the secret values that satisfy one of $C_{i_1}, C_{i_2}..C_{i_w}$. Since each clause has weight 1 and Max-SMT returns the set of clauses that maximizes the weight it follows that L induces the partitioning with the maximum number of equivalence classes and hence maximum leakage.

Algorithm 1 MaxLeak: Single-Run Side-Channel Analysis

Inputs: Program $P(h, l)$ and cost function $cost$.
Perform symbolic execution over $P(h, l)$.
Compute path conditions $PC_1(h, l), PC_2(h, l)..PC_n(h, l)$.
Compute observables $\mathcal{O} = \{cost(PC_i(h, l)) | i = 1, n\}$.
for each cost $o_i \in \mathcal{O}$ **do**
  Build Max-SMT clause with weight 1: $C_i :: (\bigvee_{cost(PC_j)=o_i} PC_j(h_j, l))$
end for
Solve $C_1, C_2..C_m$ with Max-SMT solver.
Let L be the solution returned by Max-SMT with weight w
return Low input value: L and max number of observables: w

C. Examples

Let us now analyze Example 1 using the Max-SMT formulation. The example has 5 symbolic paths, with the following (renamed) path conditions:
$PC_1 : l < 0 \wedge h_1 < 0$ with cost 1,
$PC_2 : l < 0 \wedge h_2 \geq 0 \wedge h_2 < 5$ with cost 2,
$PC_3 : l < 0 \wedge h_3 \geq 5$ with cost 3,
$PC_4 : l \geq 0 \wedge h_4 > 1$ with cost 4,
$PC_5 : l \geq 0 \wedge h_5 \leq 1$ with cost 5.

Its encoding as a Max-SMT problem is as the set of the following clauses, each with weight 1:
$C_1 :: (l < 0 \wedge h_1 < 0)$
$C_2 :: (l < 0 \wedge h_2 \geq 0 \wedge h_2 < 5)$
$C_3 :: (l < 0 \wedge h_3 \geq 5)$
$C_4 :: (l \geq 0 \wedge h_4 > 1)$
$C_5 :: (l \geq 0 \wedge h_5 \leq 1)$

Max-SMT reports that C_1, C_2, C_3 are satisfiable with $l = -1$ as a solution (Max-SMT also gives solutions for h_1, h_2, h_3, h_4, h_5 which we ignore here) resulting in maximum weight 3. Thus the low input that achieves maximum leakage is $l = -1$ and the leakage is $log_2(3) = 1.58$ bits.

Consider another example (see Figure 4). The example has 4 symbolic paths with following path conditions and costs:
$PC_1 : h_1 > 0 \wedge l < 0$ with cost 1,
$PC_2 : h_2 > 0 \wedge l \geq 0$ with cost 2,
$PC_3 : h_3 \leq 0 \wedge l < 5$ with cost 1,
$PC_4 : h_4 \leq 0 \wedge l \geq 5$ with cost 4.

Note that the paths that have conditions PC_1 and PC_3 both have same cost (1). The Max-SMT clauses are as follows, each with weight 1: $C_1 :: ((h_1 > 0 \wedge l < 0) \vee (h_3 \leq 0 \wedge l < 5))$
$C_2 :: (h_2 > 0 \wedge l \geq 0)$
$C_3 :: (h_4 \leq 0 \wedge l \geq 5)$

The result of Max-SMT solving is that for $l = 0$, the first two clauses are satisfied with maximum weight 2, hence maximum leakage is $\log_2(2) = 1$ bit.

V. Multi-run Analysis (Algorithm **MaxLeak**$_k$)

We consider now the case where a malicious agent performs a multi-run attack to gather information for deducing h or narrowing down its possible values. Such an attack consists of a *sequence* of attack steps, where each step consists in

querying the program on a chosen low input and measuring the cost of the side-channel for that program run. An attack ends if either the secret changes or if the attacker stops querying the system. We consider the case of *non-adaptive* attacks, where the attacker does not have access to the system's responses until the end of the attack. Thus, when choosing a message, she cannot take into account the outcomes of her previous queries.

Let us analyze the attacker's knowledge after k rounds of observation. Suppose that the attacker picked values L_1, L_2 .. L_k and observed the program k times, by running $P(H, L_i)$, for $i = 1 \dots k$. Suppose each $P(H, L_i)$ leads to an observation o_i. Intuitively, with each new observation, the attacker can infer more constraints on the secret, i.e., after k observations, the attacker learns that the secret is coherent with o_1 under L_1, with o_2 under L_2, .. and with o_k under L_k.

Similar to the single-run analysis, we say that two secret values H_1 and H_2 are *indistinguishable* under input sequence $\langle L_1, L_2..L_k\rangle$ if they lead to the same observable *sequence* $\langle o_1, o_2, ..o_k\rangle$.

Proposition 1: The indistinguishability relation under $\langle L_1, L_2, ..L_k\rangle$ forms an equivalence relation on the secret values.

Proof 2: Reflexivity, symmetry and transitivity are easy to check because the program is deterministic. For example for transitivity we need to show $H_1 \approx H_2$ and $H_2 \approx H_3$ implies $H_1 \approx H_3$.

$H_1 \approx H_2$ means that H_2 and H_1 under input sequence $\langle L_1, L_2..L_k\rangle$ lead to the same observable sequence s, and $H_2 \approx H_3$ means that H_2 and H_3 under input sequence $\langle L_1, L_2..L_k\rangle$ lead to the same observable sequence s'. Hence, because the program is deterministic, $s = s'$ and H_1 and H_3 under input sequence $\langle L_1, L_2..L_k\rangle$ lead to the same observable sequence s.

The approach described in the previous section generalizes naturally for a multi-run analysis, where in the case of a k-step attack, we analyze the k-composition of the program, denoted $P_k(h, l_1, l_2..l_k)$:

$$P(h, l_1); P(h, l_2); ..P(h, l_k)$$

In other words, we consider running the same program k times, with different symbolic inputs: $l_1, l_2..l_k$. Note that h remains the same across the k runs (since the secret is the same across the runs). Each path π in $P_k(h, l_1, l_2..l_k)$ is a composition of paths $\pi_1; \pi_2; ..\pi_k$, where each π_i is a path in the i-th program version, $P(h, l_i)$. We define the cost $cost_k(\pi) = \langle o_1, o_2, ..o_k\rangle$, where each o_i is the cost of π_i, i.e., an observable for the composite program is the *sequence* of k side-channel measurements made during the attack. With the new formulation of the observables, we can apply Algorithm 1 to P_k (where the low input l is now the sequence $\langle l_1, \dots, l_k\rangle$) to compute the sequence of low inputs that lead to the maximum number of different observable tuples in k steps. We denote this as $\mathbf{MaxLeak}_k$.

Theorem 2: The solution returned by Algorithm 1 for $P_k(h, l_1, l_2..l_k)$ and $cost_k$ yields maximum leakage for program P after k runs.

Theorem 2 follows directly from Theorem 1 (the program being analyzed is the k-composition of P and the cost function is $cost_k$).

Consider again Example 1 in Fig. 3. In two runs, Max-SMT returns 4 satisfiable clauses out of 13 distinct clauses. Hence, there are 4 observables and the maximum leakage is 2 bits. The low inputs are $l_1 = 0$ and $l_2 = -1$. For Example 2 in Fig. 4, in two runs, Max-SMT reports 2 satisfiable clauses out of 7 clauses. Therefore the number of observables is 2, the same as in the first run, and this is the maximum leakage possible.

We also formalize here the intuition that the attacker obtains more information about the secret with each program run. The attacker obtains more information (or equivalently reduces the uncertainty) about the secret as it refines the partition induced by the input with each program run. If two values H_1 and H_2 of the secret h are indistinguishable, then they are in the same equivalence class in the equivalence relation. When the attacker runs the program one more time, she can distinguish more values of the secret h, by splitting some equivalence classes (in other word, the equivalence relation is refined). This results in more equivalence classes (likely of smaller sizes) so the amount of information gained increases.

Proposition 2: Let w_k and w_{k+1} be the number of observables returned by Algorithm 1 for a k-run and k+1-run analysis respectively. Then $w_k \leq w_{k+1}$.

As a corollary we have that if the leakage is the same at steps k and $k+1$, then that corresponds to maximum leakage of the program [30].

A. Greedy Multi-run Analysis (Algorithm $\mathbf{GreedyLeak}_k$)

While in the previous section we have shown how to compute maximal leakage over k runs, the above algorithm doesn't guarantee a specific leakage ordering, so for example for a program with maximal leakage of 7 bits in two runs Algorithm 1 could return an assignment where 2 bits are leaked in the first run and 5 bits are leaked in the second run.

An important leakage ordering is the one corresponding to an attacker at each step picking the low input returning the maximal leakage for that run. We can easily adapt our previous algorithm to capture this attacker. To do so we should proceed as follows: find the maximal L_1 over one run using Algorithm 1: consider then the maximal low input over two runs where we have "blocked" the clauses satisfied by L_1, we have now L_1, L_2 and we repeat until we have found $L_1, \dots, L_k$.

The above algorithm is greedy, so it doesn't necessarily return the sequence with the maximal possible leakage at each step. For example consider the case where L_1 and L_1' both return the same maximal leakage in the first run but after L_1 the next maximal L_2 returns a higher maximal leakage than the maximal leakage that L_2' returns after L_1'. Max-SMT

could choose L_1' over L_1 and so the sequence chosen by the algorithm starting with $L_1', L_2', \ldots$ would have a lower leakage in the second element that the sequence starting with $L_1, L_2, \ldots$. We call this approach **GreedyLeak**$_k$.

Other possible attackers can be modelled in a similar fashion, e.g. an adaptive attacker that chooses the next low input following the observables from the previous rounds. In this case the attacker can be modelled as a function from sequence of observables (the history) to low inputs (the next input).

B. Information-theoretic metrics

Let us consider a generic function $\mathcal{L}$ measuring leakage in bits, e.g. channel capacity or Shannon entropy, and suppose $\mathcal{L}$ is capable to measure leakage over multiple runs of a program P given low inputs l_i. Let's denote $\mathcal{L}(P(H, L_1, \ldots, L_k))$ the leakage over k runs with low inputs $L_1, \ldots, L_k$.

We can then define the *information gain* at the $k+1$ run according to $\mathcal{L}$ and low inputs $L_1, \ldots, L_k, L_{k+1}$ as $\mathcal{L}(P_{k+1}(H, L_1, \ldots, L_{k+1})) - \mathcal{L}(P_k(H, L_1, \ldots, L_k))$, that is the difference between the leakage after $k+1$ and k runs: this is the leakage revealed by the $k+1$ run of the program P as measured by $\mathcal{L}$. It is the secret revealed by $P(H, L_{k+1})$ which had not been previously been revealed by $(P_j(H, L_1, \ldots, L_j))_{1 \leq j \leq k}$

The *remaining secrecy* at the k-th run according to $\mathcal{L}$ is defined as the difference between the initial secret and the leakage according to $\mathcal{L}$ as measured after k runs: this is the secret that has not been leaked in the k runs.

In the present work we concentrated on channel capacity as the leakage measure $\mathcal{L}$, that is the maximal leakage as measured according to Shannon entropy or Smith's min entropy [9]. In this setting we have that the information gain at the $k+1$ run is the maximal leakage revealed by the $k+1$ run as measured by Shannon or Min entropy. The remaining secrecy at the k-th run is the secret that cannot have been leaked in the k runs. This is the minimum amount of secrecy guaranteed after k runs.

If we were to take as measure of leakage $\mathcal{L}$ Shannon entropy, computed for example with model counting over the constraints as outlined in section III-B, then the information gain at the first step reduces to just the leakage (i.e. $\mathcal{H}(P) - 0 = \mathcal{H}(P) = \mathcal{L}(P)$) and the remaining secrecy at the 1st step is just the remaining uncertainty, i.e. the posterior entropy

$$\mathcal{H}(h) - \mathcal{L}(P) = \mathcal{H}(h) - (\mathcal{H}(h) - \mathcal{H}(h|P)) = \mathcal{H}(h|P)$$

The above generalize to k runs, e.g. the remaining secrecy at the k-th run would then be the remaining uncertainty about the secret given k runs and the information gain at the $k+1$ run is the reduction in uncertainty about the secret induced by the $k+1$ run of the program. As Shannon entropy is, when we know the distribution of the secret, a more precise measure of leakage than channel capacity, the induced measures of information gain and remaining secrecy will also be more precise. For example since channel capacity can be a big overestimation of leakage this will make remaining secrecy according to channel capacity a big underestimation of the non leaked secret. As a concrete example consider a password check over 1000 elements: the remaining secrecy according to channel capacity after one step will be $\log(1000) - 1 \simeq 8.9$ whereas the real remaining secrecy is closer to 9.9 bits.

Notice that in terms of our algorithm an information gain of 0 at $k+1$ means that the number of equivalence classes (maximal number of satisfied clauses) at round k and $k+1$ is equal, i.e. nothing new about the secret can be revealed at round $k+1$. A particular case of information gain being 0 is when the maximal number of satisfied clauses is d where d is the size of the secret. In that case all secret has been revealed.

Note that our algorithms find the low inputs that maximize the number of equivalence classes. Our tool can compute the sizes of each partition (using model counting) to calculate various measures such as Shannon entropy. Finding the input that maximizes entropy for a specific distribution is future work.

VI. Discussion

A. Multi-threading

So far we assumed that the program under analysis is sequential. We discuss here how to extend the analysis to multi-threaded programs using *maximal* linear schedules (i.e. thread interleavings) [7]. Suppose we have no knowledge about the next-choice distribution or the specific scheduling policy for the thread scheduler.

Intuitively each thread scheduling induces a sequential program on which we can apply the leakage analysis as described above. As there is a finite number of schedules for the current exploration depths, we can therefore compute the low input and maximum leakage for each one of them and report the worst case to the user.

To account for multi-threading we need to extend the definition of symbolic execution as well as the attacker model, since an input may result in multiple program executions, corresponding to different thread schedules. We replace IP with a set of pairs (t_i, IP_i), where each t_i identifies an active thread and IP_i represents the next instruction to be executed by thread t_i. A schedule $\mathcal{S}$ is a sequence $t_i, t_j, ...t_k$ defining the order of access to the CPU for all the active threads. The result of such an execution is a set of pairs $(\mathcal{S}_i, PC_i)$ where PC_i are path conditions and $\mathcal{S}_i$ is the schedule associated with the specific execution. We can record the schedules produced by symbolic execution into a prefix tree and define the maximal schedules:

Definition 1: A thread schedule is *maximal* if it is not a prefix of any other schedule in the paths reported by a (bounded) symbolic execution.

For a thread schedule $\mathcal{S}$ that is maximal, we define a set of path conditions

$$\Pi_{\mathcal{S}} = \{PC_i | \mathcal{S}_i \in prefix(\mathcal{S})\}$$

where $prefix(\mathcal{S})$ is the set of all the prefixes of $\mathcal{S}$ including $\mathcal{S}$. Intuitively $\Pi_{\mathcal{S}}$ contains all the path conditions for the paths

that "follow" the same schedule $\mathcal{S}$, accounting for possible early termination.

For a maximal schedule the path conditions corresponding to $\Pi_{\mathcal{S}}$ cover the entire domain input [7]. Thus, $\Pi_{\mathcal{S}}$ is the result of symbolically executing program $P(h, l)$ restricted to schedule $\mathcal{S}$ which can be seen as executing a sequential program $P_{\mathcal{S}}(h, l)$. We can therefore compute the value of low that maximizes leakage by applying Algorithm 1 to $P_{\mathcal{S}}(h, l)$ (essentially the path conditions computed by Algorithm 1 will be the path conditions corresponding to each path in $\Pi_{\mathcal{S}}$). Maximal schedules can then be ordered according to their worst leakage and the worst result is reported to the user. The approach extends to multi-run analysis as well.

As an example, consider two threads:

$T_1 :: \texttt{example1}(l, h)$;

$T_2 :: l = -l$;

As before, `h` and `l` are inputs. Method `example1` is described in Figure 3. Assume for simplicity that thread 1 invokes method `example1` atomically (e.g. in a synchronized block). Then `l=-1` gives 3 observables for thread schedule "T_1;T_2" but only 2 observables for "T_2;T_1". Thus, there may be more than one program execution associated with an input, each with different observations, and only some executions give the worst leakage.

Note that the number of possible thread interleavings may be very large and ennumerating all of them may be infeasible. The problem can be addressed by partial order reduction (POR), supported by Java Pathfinder. POR exploits the commutativity of concurrently executed instructions, which result in the same state when executed in different orders.

Note also that the non-determinism introduced by multi-threading produces some "noise" that makes the leakage smaller [31], [32] and this is not captured by our analysis. However, if the attacker is allowed to observe the scheduling sequence, the leakage increases as in the example. Furthermore, our analysis produces a thread schedule which can be analyzed by developers for debugging and can thus be quite useful in practice.

Further there are more powerful notions of tree-like [33] and probabilistic schedulers [31], [32] that would allow us to compute more precise information on the leakage. We plan to explore them in depth for future work.

B. Garbage Collection

The view of the adversaries as defined in this paper disregards the memory management handled by garbage collection whose execution is unpredictable and may affect the side channel measurements. Our analysis can be adapted to more refined cost models and different garbage collection policies by modifying and controlling the scheduler and the garbage collection inside JPF's custom JVM.

Nondeterminism and multi-threading also provide a powerful mechanism for studying the effects of the garbage collection, whose role is to find the unnecessary (garbage) objects and to remove them. We can view the garbage collector as a separate thread that interferes with the execution of the program analysis, affecting both its execution time and memory execution costs. Our tool implements a custom VM which allows to experiment with different garbage collection policies. We leave this for future work.

```
boolean check(byte[] secret, byte[] input){
  for (int i = 0; i < SIZE; i++){
    if (secret[i] != input[i]){
        return false;
    }
    Thread.sleep(25L);
  }
  return true;
}
```

Fig. 5. Password Check

VII. Implementation and Experience

We implemented our analysis in Symbolic PathFinder [23]. We use Z3 [18] (bit-vector theory) for SMT and Max-SMT solving. We implemented JPF listeners to monitor the bytecode instructions executed by the program, and to perform the analysis on the following possible type of side-channels: timing (by measuring execution time of each instruction according to a cost model), memory-usage (by computing the number of live heap objects along a path), network and file communication (by providing models for network and file interactions and computing number of bytes written to an output stream or file via methods `write` of `java.io.OutputStream` and `java.io.FileOutputStream` respectively). For experiments we used a simple timing model that allows easy comparison with ACSAC.

We evaluated our implementation based on two sets of experiments.

- We compared the *single-run analysis* (i.e. Algorithm MaxLeak, various configurations) with another symbolic approach, ACSAC [6] (described in detail below) and with brute-force enumeration (the baseline for our approach).
- We also evaluated the *multi-run analysis* (MaxLeak_k) for increasing number of k. We compared the "full approach" MaxLeak_k, which computes the input sequence all at once using Max-SMT solving, with the "greedy" approach GreedyLeak (described in Section V-A) which computes the inputs one by one.

We analyzed a set of Java examples for the different side channels described above. Here we describe in detail the analysis of two representative examples: password checking and cryptographic functions (which we could convert easily into C allowing a comparison with ACSAC).

All the experiments were run on a standard MacBook Pro with 2.2 GHz Intel Core i7 and 16 GB 1600 MHz DDR3.

A. Examples

Timing channel in password check Fig. 5 shows the code of a simple segmented password checking program. Here

`secret` represents the "high" value and `input` is the "low" value; both `secret` and `input` have the same `SIZE` (i.e. the same length as strings). The code checks the password and the input character by character and returns false as soon a mismatching character is found. This implementation is insecure against an attacker measuring the time taken by the method to return "true" or "false". We analyzed this example for different values of `SIZE` and element range.

Timing channel in cryptographic functions Fig. 6 shows the implementation of fast modular exponentiation [27] – an operation that is typically found in asymmetric cryptographic algorithms such as RSA used by modern computers to encrypt and decrypt messages. Here `e` is the secret, `num` is the public input, and `m` is a constant (the product of two prime numbers). Method `modPow1` iterates over the bits in the secret `e` (the `while` loop) and performs different computations based on whether the bit is 1 or 0. The timing channels in this kind of functions result from the fact that modular multiplication is *not* constant time: for some operands it takes longer than for others (because a so-called extra reduction step). With the reduction step one gets a subtler dependency between low input `num`, key `e`, and timing. By varying the low input, an attacker can in some circumstances guess the key [34]. To study this phenomenon, we modified the algorithm to include a simple reduction step (see Fig. 7; second `if` statement has a dependency on low input that can be exploited for an attack). Other more involved methods are treated similarly. We analyzed the program for different values of the modulo `m` (where `num` is bound by `m`) and different `e` lengths.

While the password check involves only simple computations, the analysis of modular exponentiation results in non-linear constraints that are hard to solve with existing constraint solvers, and hence it is good to stress-test our proposed technique.

B. MaxLeak (default) vs. MaxLeak (No Solver)

We note that in our approach there is some redundancy between the solving performed by SPF and the solving performed by Max-SMT. We therefore compared MaxLeak with two configurations: running SPF with and without the solver. With the first option (default), SPF uses constraint solving to rule out infeasible PCs, so only the feasible PCs are used in the Max-SMT calculation. With the second option (No Solver), SPF performs no solving, and collects all the PCs (including the infeasible ones) and sends them to Max-SMT for solving (which implicitly also rules out infeasible PCs).

C. The ACSAC [6] approach

Most of the previous work on automated QIF (Section VIII) performs the analysis assuming the "low" input is given. While this is a sound assumption, since the low input is controlled or at least is known by the attacker, those techniques cannot synthesize the low inputs that maximize the leakage. An exception is the work in [6]: the technique is capable, in the case of a single run, to determine if the leakage is at least k bits, and provide a low input that makes the program leak k bits. As such ACSAC [6] is a good candidate to benchmark the performance of our approach for the single run analysis. At a high level, ACSAC computes the self-composition of k copies of the program, and computes an assertion that these k copies cannot create k different observables. The bounded model checker CBMC [35] is then used to verify this assertion.

To perform the comparison with ACSAC we manually translated the programs in C code (the input language of CBMC) and instrumented the code by adding a `time` variable to simulate an adversary observing the timing channel (as described in Section III). The values of the `time` variable at the end of the program constitute the "observables" for the ACSAC approach. Note also that CBMC uses a different solver than Z3, which is better for bitvectors [35].

For ACSAC we report the time it takes in the last step to validate the assertion (note that in reality ACSAC performs an iterative approach, for an increasing number of observables, until the assertion becomes valid, so the overall time is higher than the one we report).

D. Results and Discussion

The results of the single-run experiments are shown in Figures 8 and 9, while the results for the multi-run experiments are shown in Figures 10 and 11.

In the tables, k is the number of steps (for the multi-run analysis), maxObs is the maximum number of observables reported by Max-SMT, #PCs is the number of PCs reported by SPF, #Obs is the number of clauses, time SPF is time to run SPF in seconds, and time Max-SMT is the time to run Max-SMT in seconds. A "–" means analysis timed out in 1 hour.

For single-run, MaxLeak (No Solver) performs better than MaxLeak(default). The time for running SPF is much smaller (since it uses no solving) but the number of generated paths may be very large (since it also includes infeasible paths) up to the point that Max-SMT can no longer solve the generated clauses (last line in Figure 9).

The results also show that ACSAC does not scale well for the single-run analysis[1]. The reason is that for a single run, ACSAC requires the composition of $NObs$ copies of the program to validate the assertion (where $NObs$ is the number of possible observations in one run). In contrast, MaxLeak uses only one copy of the program for the single-run analysis. Thus, although we use different tools and different solvers for the comparison, we believe there is an inherent complexity problem with the ACSAC approach (supported by our experiments).

For multi-run analysis, the results show, as expected, that MaxLeak is more expensive than GreedyLeak, both in the number of PCs generated and the analysis time (where the Max-SMT solving time is dominant). On the other hand GreedyLeak does not always returns the maximum leakage: Fig. 11 shows the number of observables returned by

[1]The time-out at small configurations in Figure 9 may be due to some hard to solve constraints that are unsat at smaller configurations but become sat when `m` is larger.

```
int modPow1(int num, int e, int m){
  int s = 1, y = num, res=0;
  while (e > 0) {
    if (e % 2 == 1) {
      res = (s * y) % m;
    } else {
      res=s;
    }
    s = (res * res) % m;
    e /= 2;
  }
  return res;
}
```

Fig. 6. Modular Exponentiation

```
int modPow2(int num, int e, int m){
  int s = 1, y = num, res=0;
  while (e > 0) {
    if (e % 2 == 1) {
      //reduction:
      int tmp = s * y;
      if (tmp > m){
        tmp = tmp - m;

      }
      res = tmp % m;
    } else {
      res=s;
    }
    s = (res * res) % m;
    e /= 2;
  }
  return res;
}
```

Fig. 7. Simple Reduction

SIZE	maxObs	MaxLeak (default)				MaxLeak (No solver)				ACSAC
		#PC	#Obs	time SPF	time Max-SMT	#PC	#Obs	time SPF	time Max-SMT	
10	11	11	11	0.561	0.046	11	11	0.333	0.047	2m33.349
50	51	51	51	8.662	4.958	51	51	0.566	5.07	-
100	101	101	101	59.852	36.061	101	101	0.932	36.983	-
200	201	201	201	7m50.156	16m29.761	201	201	3.086	16m37.846	-

Fig. 8. Single-run analysis of password check (range 1..62). Brute force times out in all configurations

Modulo	Len	maxObs	MaxLeak (default)				MaxLeak (No solver)				ACSAC	BruteForce
			#PC	#Obs	time SPF	time Max-SMT	#PC	#Obs	time SPF	time Max-SMT		
1717	3	6	13	6	8.830	0.483	40	9	0.417	0.763	1m22.537	0.103
	4	9	38	9	1m9.719	1.675	121	12	0.569	4.264	-	0.099
	5	12	107	12	6m10.376	7.585	364	15	0.899	27.448	-	0.097
	6	15	285	15	27m26.593	34.665	1093	18	1.967	3m59.985	-	0.110
	7	18	-	-	-	-	3280	21	6.367	43m5.840	-	0.122
834443	3	6	13	6	4.810	0.398	40	9	0.421	0.811	-	0.328
	4	9	40	9	24.222	2.151	121	12	0.557	5.491	-	0.661
	5	12	121	12	1m59.615	10.387	364	15	0.866	25.362	-	1.621
	6	15	364	15	8m50.549	1m12.780	1093	18	1.997	3m50.015	-	4.402
	7	18	1093	18	37m49.129	6m6.093	3280	21	6.487	41m20.159	-	9.739
1964903306	3	6	13	6	5.138	0.509	40	9	0.436	1.010	5.211	6m52.263
	4	9	40	9	50.067	3.119	121	12	0.604	5.525	1m5.912	18m19.167
	5	12	121	12	7m58.086	1m5.797	364	15	0.980	1m0.471	10m47.589	47m18.512
	6	15	-	-	-	-	1093	18	2.002	31m42.575	-	-
	7	18		-	-	-	3280	21	8.455	-	-	-

Fig. 9. Single-run analysis of `modPow2` (Len is `e`'s length in bits).

GreedyLeak vs MaxLeak or brute force, when they can complete. Adding a backtracking mechanism to GreedyLeak could address the problem.

The "no solver" option performs well for the password check but for `modPow` this option generates so many paths that we could not obtain any result from Max-SMT. This may be due to some bottleneck in the front-end of the solver which we hope to solve in the future. We are talking to the Z3 developers to explore a better integration between the Z3 solving on the SPF side and Max-SMT.

As expected, the brute-force approach performs well for small configurations, but it becomes intractable for larger sizes. In our experiments with modulo exponentiation, brute force did outperform MaxLeak in single-run analysis for small values of `m`. However, for a large `m`, which is often the case in cryptographic systems, MaxLeak outperformed brute force. This is even clearer in the multi-run analysis when brute force failed to synthesize any 3-run attacks, even for a small value of `m`.

Nevertheless, all compared approaches suffer from scalability issues as the length of the secret increases. However, we noticed that the experiments expose some regularity in the results. For example, for single-run, the maximum number of observables for the password check is $SIZE + 1$ while for

RANGE	SIZE	k	MaxLeak$_k$(No solver)					GreedyLeak$_k$(No solver)				
			#PC	#Obs	maxObs	time SPF	time Max-SMT	#PC	#Obs	maxObs	time SPF	time Max-SMT
2	2	2	9	9	4	0.652	0.042	9	9	4	0.104	0.030
	3	2	16	16	6	0.685	0.115	16	16	6	0.113	0.088
		3	64	64	7	0.736	2.797	64	64	7	0.151	1.154
		4	256	256	8	1.008	1m33.259	256	256	8	0.202	15.425
	4	2	25	25	8	0.695	0.382	25	25	8	0.154	0.306
		3	125	125	10	0.884	18.147	125	125	10	0.211	9.495
		4	625	625	12	1.156	17m41.007	625	625	12	0.404	3m28.896
		5	3125	3125	-	2.555	-	3125	3125	-	1.321	-
3	2	2	9	9	5	0.680	0.036	9	9	5	0.099	0.031
		3	27	27	6	0.693	0.272	27	27	6	0.114	0.140
		4	81	81	7	0.809	4.255	81	81	7	0.130	0.593
		5	243	243	8	0.937	1m5.058	243	243	8	0.182	4.867
		6	729	729	9	1.229	13m9.257	729	729	9	0.307	57.014
	3	2	16	16	7	0.678	0.115	16	16	7	0.111	0.088
		3	64	64	9	0.761	2.464	64	64	9	0.147	1.304
		4	256	256	11	0.951	1m6.886	256	256	11	0.222	28.445
		5	1024	1024	13	1.400	33m31.888	1024	1024	13	0.441	5m52.209
		6	4096	4096	-	2.837	-	4096	4096	-	1.417	-

Fig. 10. Multi-run analysis of password check. Brute force takes a couple of seconds.

Modulo	Len	k	MaxLeak$_k$(default)					GreedyLeak$_k$(default)					BruteForce	
			#PC	#Obs	maxObs	time SPF	time Max-SMT	#PC	#Obs	maxObs	time SPF	time Max-SMT	maxObs	time
1717	3	2	31	14	7	1m49.327	6.101	13	10	7	11.072	3.901	7	0.144
	4	2	128	29	15	17m28.882	47.972	38	19	14	1m15.528	6.614	15	4.792
		3						38	30	15	1m46.254	18.051		
	5	2	-	-	-	-	-	107	31	24	8m11.855	13.077	27	11.996
		3	-	-	-	-	-	107	64	29	9m32.251	1m16.879	-	-
		4	-	-	-	-	-	107	75	31	9m49.246	1m16.903	-	-
	6	2	-	-	-	-	-	285	46	35	38m05.420	51.766	40	28.820
		3	-	-	-	-	-	285	108	52	41m14.786	3m34.222	-	-
		4	-	-	-	-	-	285	157	59	43m48.355	3m36.526	-	-
		5	-	-	-	-	-	285	178	63	46m08.591	3m55.498	-	-
834443	3	2	31	14	7	40.518	2.067	13	11	7	8.078	0.384	-	-
	4	2	156	29	15	7m04.709	36m49.891	40	18	14	45.272	11.351	-	-
		3						40	26	15	1m01.474	4.973	-	-
	5	2	-	-	-	-	-	121	32	26	3m42.725	1m53.156	-	-
		3	-	-	-	-	-	121	58	-	4m34.153	-	-	-
	6	2	-	-	-	-	-	364	55	-	15m20.872	-	-	-
1964903306	3	2	31	14	7	47.246	2.945	13	9	7	8.276	1.183	-	-
	4	2	156	29	-	15m29.818	-	40	20	14	1m13.667	32.551	-	-
		3	-	-	-	-	-	40	29	-	1m15.041	-	-	-
	5	2	-	-	-	-	-	121	31	-	12m18.071	-	-	-

Fig. 11. Multi-run analysis of `modPow2`.

`modPow` it is $3 * (Length - 1)$. This suggests that one can extrapolate from these results and get the vulnerability for larger input configurations, even if they can not be analyzed directly using symbolic execution. We plan to investigate this further.

Note also that the password check is a special case where the choice of the low input in a single run is irrelevant. Therefore the results for a single run are not very illuminating. However the results (both for single- and multi-run analysis) do support the fact that our symbolic techniques perform well when only simple linear constraints are involved, and can potentially scale to large programs. Note also that in the case of linear constraints, other solvers can perform much better than Z3bitvector. However, solvers struggle in the presence of non-linear constraints (as for the `modPow` example).

We also performed some preliminary experiments with an *adaptive* greedy approach (see Section V-A) which creates a new Max-SMT problem with each new observation made. Our preliminary results indicate that the adaptive approach could scale much better than the non-adaptive one: although we need to solve more Max-SMT problems than in the non-adaptive case, each problem is smaller and therefore easier to solve. Furthermore adaptive strategies may result in smaller "attack" sequences and may be more appropriate for examples such as the password check (e.g. to model an attack that guesses the password character by character). We plan to explore this topic in depth in the future.

E. Security relevance of experiments

Let us give some intuition on what these experiments mean from a security point of view.

Consider the password experiments shown in Figure 10 and consider for example the case: range 2, size 3, i.e. we have a 3 bits secret. For 3 runs (k=3) the results indicate 7 maxObs hence the channel capacity is $\log_2(7) \simeq 2.8$ bits which means that all the secret is at most leaked in the three runs. It is easy to see that this is a tight bound.

Similarly, consider the modulus exponentiation in Figure 11 with modulus 1717 and a 6 bits secret; then in 5 runs using GreedyLeak we found 63 maxObs i.e. at most 5.9(=$\log_2(63)$) bits can be leaked, which is basically all the secret. This is also a tight bound, which is rather surprising by looking at the code.

It is in fact easy to demonstrate that the bounds obtained by our analysis are tight: by proposition 1 the indistinguishability relation under $\langle L_1, L_2, ..L_k \rangle$ forms an equivalence relation on the secret values, and each observable represents an equivalence class on the secret. There are 63 observables and 63 secrets (this is because the secret cannot take the value 0), which implies each class has only one element, which means that each observables is associated to a unique secret, and so *all* the secret is leaked.

Note also that the GreedyLeak$_k$ approach is capable of obtaining tight bounds on the leakage, but, as expected, the attack sequence may be larger than for MaxLeak$_k$. For example, in Figure 11, modulus 1717 and password length 4 bits, the greedy approach gives 3 runs to obtain the maximum number of observables (15) while the same information can be obtained only with two runs (shown with MaxLeak$_k$).

Finally note that Max-SMT also generates the concrete solutions for the low input sequence, showing the developers how to precisely obtain the program executions that lead to maximum leakage.

VIII. Related Work

Side-channel attacks have been well studied in the literature [10], [1], [27] however there are few automated approaches that are able to quantify information leakage over multiple runs [10], [36].

Köpf and Basin [10] developed an automated approach for multi-run analysis in a more general setting, addressing adaptive attacks. However the technique is based on an enumeration algorithm (doubly-exponential in the number of attack steps); they also present a greedy heuristic to compute the remaining entropy of the secret after k steps in the context of adaptive attacks. We propose here a symbolic approach that leverages Max-SMT solving to avoid an explicit search of best attacks. Our experiments show the technique has merits compared with brute-force enumeration, indicating its potential for synthesizing adaptive attacks as well. We leave this for future work.

Heusser and Malacaria [6] use the model checker CBMC to determine the public input that results in maximal leakage in the more general context of quantitative flow analysis. The technique requires an n-step composition of the program, where n is the number of different observations one can make in one run of the program, and does not address multi-run analysis. Our technique can determine the same information from the analysis of only one program run but it is limited to the specific context of side-channel analysis. We showed experimentally that the Max-SMT approach can be much better in practice.

Backes et al. [37] use symbolic techniques and model counting for quantitative information flow analysis but they assume a-priori knowledge of the public inputs. Previous work [12], [15], [16] has used symbolic execution for quantitative flow analysis in the simpler context of single-run attacks where only upper-bounds on the leakage are computed. That work does not compute the public user input that maximizes the leakage, it does not address multi-run attacks and it does not use probabilistic symbolic execution for computing information theoretic metrics – all our contributions here.

Mardziel et al. [36] generalizes the work of Köpf and Basin [10] by considering probabilistic systems to account for secrets that change over time. They use probabilistic programming to implement a model of information flow for probabilistic, interactive systems with adaptive adversaries and to compute the leakage. This suggests a possible connection with the probabilistic capabilities of SPF that we plan to explore in future work.

Cache side channels are studied extensively in [13], [14] although not in a multi-run setting. Our tool is built on top of

JPF, a custom VM with its own memory model, and can thus analyze some memory side-channels but we do not currently model architecturespecific cacheside channels as in [13], [14] which we leave for future work.

IX. Conclusion

We described a symbolic execution approach to side-channels detection and quantification. We defined a new application of Max-SMT solving to identify the low input triggering the most vulnerable behavior of the program and analyze leakage of multi-run attacks. We implemented the analysis in Symbolic PathFinder and showed its merit on several examples.

We made few attempts to optimizing our implementation and at present it does not scale when the number of time steps is large. However we note that the research area of Max-SMT solving is still very young and we are optimistic that it will increase in the future attaining the level of maturity of SMT solving that has grown explosively in the past decade. Further our analysis may benefit from a tighter integration between the symbolic exploration and the Max-SMT solving and from distributing the analysis, e.g. by creating parallel versions of the symbolic execution engine and creating a new parallel job with each new observation made. Other areas for future work include investigating the use of qCoral [8] for quantifying solution spaces over non-linear constraints and extending our analysis for leakage computation in the presence of noisy observations.

Acknowledgment

We are grateful to Nikolaj Bjorner for answering many of our questions about Z3 and Max-SMT. We also thank the anonymous reviewers for their constructive comments. This material is based on research sponsored by DARPA under agreement number FA8750-15-2-0087. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. Malacaria research was supported by EPSRC grant EP/K032011/1.

References

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

[2] J. Kelsey, "Compression and information leakage of plaintext," in *Revised Papers from the 9th International Workshop on Fast Software Encryption*, FSE '02, (London, UK, UK), pp. 263–276, Springer-Verlag, 2002.

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

[4] P. Malacaria, "Assessing security threats of looping constructs," in *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '07, (New York, NY, USA), pp. 225–235, ACM, 2007.

[5] R. Nieuwenhuis and A. Oliveras, "On SAT Modulo Theories and Optimization Problems," in *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing*, SAT'06, (Berlin, Heidelberg), pp. 156–169, Springer-Verlag, 2006.

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

[7] 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, (Piscataway, NJ, USA), pp. 622–631, IEEE Press, 2013.

[8] M. Borges, A. Filieri, M. d'Amorim, C. S. Păsăreanu, and W. Visser, "Compositional Solution Space Quantification for Probabilistic Software Analysis," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, (New York, NY, USA), pp. 123–132, ACM, 2014.

[9] G. Smith, "On the Foundations of Quantitative Information Flow," in *Proceedings of the 12th International Conference on Foundations of Software Science and Computational Structures*, FOSSACS '09, (Berlin, Heidelberg), pp. 288–302, Springer-Verlag, 2009.

[10] B. Köpf and D. Basin, "An Information-theoretic Model for Adaptive Side-channel Attacks," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, (New York, NY, USA), pp. 286–296, ACM, 2007.

[11] P. Malacaria and H. Chen, "Lagrange multipliers and maximum information leakage in different observational models," in *Proceedings of the third ACM SIGPLAN workshop on Programming languages and analysis for security*, PLAS '08, (New York, NY, USA), pp. 135–146, ACM, 2008.

[12] Q.-S. Phan, P. Malacaria, O. Tkachuk, and C. S. Păsăreanu, "Symbolic Quantitative Information Flow," *SIGSOFT Softw. Eng. Notes*, vol. 37, pp. 1–5, Nov. 2012.

[13] B. Köpf, L. Mauborgne, and M. Ochoa, "Automatic quantification of cache side-channels," in *Proceedings of the 24th international conference on Computer Aided Verification*, CAV'12, (Berlin, Heidelberg), pp. 564–580, Springer-Verlag, 2012.

[14] G. Doychev, D. Feld, B. Köpf, L. Mauborgne, and J. Reineke, "CacheAudit: A Tool for the Static Analysis of Cache Side Channels," in *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, (Berkeley, CA, USA), pp. 431–446, USENIX Association, 2013.

[15] 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, (New York, NY, USA), pp. 283–292, ACM, 2014.

[16] 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, (New York, NY, USA), pp. 105–108, ACM, 2014.

[17] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, pp. 385–394, July 1976.

[18] L. De Moura and N. Bjørner, "Z3: an efficient SMT solver," in *Proceedings of the 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08, (Berlin, Heidelberg), pp. 337–340, Springer-Verlag, 2008.

[19] E. Bounimova, P. Godefroid, and D. Molnar, "Billions and Billions of Constraints: Whitebox Fuzz Testing in Production," in *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, (Piscataway, NJ, USA), pp. 122–131, IEEE Press, 2013.

[20] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley, "Enhancing Symbolic Execution with Veritesting," in *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, (New York, NY, USA), pp. 1083–1094, ACM, 2014.

[21] L. Ciortea, C. Zamfir, S. Bucur, V. Chipounov, and G. Candea, "Cloud9: A Software Testing Service," *SIGOPS Oper. Syst. Rev.*, vol. 43, pp. 5–10, Jan. 2010.

[22] "Java PathFinder." http://babelfish.arc.nasa.gov/trac/jpf/.

[23] 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*, pp. 1–35, 2013.

[24] J. A. D. Loera, R. Hemmecke, J. Tauzer, and R. Yoshida, "Effective lattice point counting in rational convex polytopes," *Journal of Symbolic Computation*, vol. 38, no. 4, pp. 1273 – 1302, 2004. Symbolic Computation in Algebra and Geometry.

[25] L. De Moura and N. Bjørner, "Satisfiability modulo theories: introduction and applications," *Commun. ACM*, vol. 54, pp. 69–77, Sept. 2011.

[26] R. Karp, "Reducibility among combinatorial problems," in *Complexity of Computer Computations* (R. Miller, J. Thatcher, and J. Bohlinger, eds.), The IBM Research Symposia Series, pp. 85–103, Springer US, 1972.

[27] 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, (London, UK, UK), pp. 104–113, Springer-Verlag, 1996.

[28] A. Darvas, R. Hähnle, and D. Sands, "A theorem proving approach to analysis of secure information flow," in *Proceedings of the Second international conference on Security in Pervasive Computing*, SPC'05, (Berlin, Heidelberg), pp. 193–209, Springer-Verlag, 2005.

[29] G. Barthe, P. R. D'Argenio, and T. Rezk, "Secure Information Flow by Self-Composition," in *Proceedings of the 17th IEEE workshop on Computer Security Foundations*, CSFW '04, (Washington, DC, USA), IEEE Computer Society, 2004.

[30] P. Malacaria, "Algebraic foundations for quantitative information flow," *Mathematical Structures in Computer Science*, vol. 25, pp. 404–428, 2 2015.

[31] H. Chen and P. Malacaria, "The optimum leakage principle for analyzing multi-threaded programs," in *Information Theoretic Security* (K. Kurosawa, ed.), vol. 5973 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 2010.

[32] H. Chen and P. Malacaria, "Quantitative Analysis of Leakage for Multi-threaded Programs," in *Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security*, PLAS '07, (New York, NY, USA), pp. 31–40, ACM, 2007.

[33] K. S. Luckow, C. S. Pasareanu, M. B. Dwyer, A. Filieri, and W. Visser, "Exact and approximate probabilistic symbolic execution for nondeterministic programs," in *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, pp. 575–586, 2014.

[34] J.-F. Dhem, F. Koeune, P.-A. Leroux, P. Mestr, J.-J. Quisquater, and J.-L. Willems, "A practical implementation of the timing attack," in *Smart Card Research and Applications* (J.-J. Quisquater and B. Schneier, eds.), vol. 1820 of *Lecture Notes in Computer Science*, pp. 167–182, Springer Berlin Heidelberg, 2000.

[35] "CBMC." http://www.cprover.org/cbmc/.

[36] 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*, pp. 540–555, 2014.

[37] 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, (Washington, DC, USA), pp. 141–153, IEEE Computer Society, 2009.