Online Synthesis of Adaptive Side-Channel Attacks Based On Noisy Observations

Lucas Bang, Nicolás Rosner, and Tevfik Bultan
Department of Computer Science
University of California Santa Barbara
{*bang, rosner, bultan*}*@cs.ucsb.edu*

***Abstract*—We present an automated technique for synthesizing adaptive attacks to extract information from program functions that leak secret data through a side channel. We synthesize attack steps dynamically and consider noisy program environments. Our approach consists of an offline profiling phase using symbolic execution, witness generation, and profiling to construct a noise model. During our online attack synthesis phase, we use weighted model counting and numeric optimization to automatically synthesize attack inputs. We experimentally evaluate the effectiveness of our approach on DARPA benchmark programs created for testing side-channel analysis techniques.**

We present an automated technique for synthesizing adaptive attacks to extract information from program functions that leak secret data through a side channel. We synthesize attack steps dynamically and consider noisy program environments. Our approach consists of an offline profiling phase using symbolic execution, witness generation, and profiling to construct a noise model. During our online attack synthesis phase, we use weighted model counting and numeric optimization to automatically synthesize attack inputs. We experimentally evaluate the effectiveness of our approach on DARPA benchmark programs created for testing side-channel analysis techniques.

1. Introduction

Side-channel attacks are an important source of software vulnerabilities. By measuring a software system's processing time, power usage, or memory, a malicious adversary can gain information about private data. For instance, exploitable timing channel information flows were discovered for Google's Keyczar Library [23], the Xbox 360 [1], implementations of RSA encryption [8], and the open authorization protocol OAuth [2]. These vulnerabilities highlight the need for preemptively discovering of the possibility of side-channel attacks and their removal from software.

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. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

In this paper, we present a method for automatically and dynamically synthesizing adaptive side-channel attacks under noisy observations against code segments that manipulate secret data. To illustrate the main idea, consider the following scenario. An adversary obtains the program source code and system specification for a server application, but does not have access to private values stored on the server on which the application runs. We show that by analyzing the source code and performing offline profiling on a mock server under his control, an adversary can develop a probabilistic model of the relationship between the inputs, side-channel measurements, and secret values. Using this model, the adversary can mount a side-channel attack against the real platform, adaptively selecting inputs that leak secret values.

In this work, we show how to automate the type of attack just described. We give a solution for the problem based on information theory and Bayesian inference. We use state-of-the-art tools and techniques including symbolic execution, symbolic model-counting, and numeric optimization. We implemented our approach, targeting networked Java server applications, and experimentally show that it works on a set of benchmarks by either synthesizing an attack or by reporting that an attack was not found.

There has been prior work: on secure information flow providing methods for detecting insecure flows [35]; on quantitative information flow presenting techniques for measuring the amount of information leaked through indirect flows [38]; and on analysis of adaptive side-channel adversaries providing techniques for automatically reasoning about malicious users [22]. However, despite influential prior work in these areas, existing adaptive adversary models for reasoning about malicious users (1) rely on explicit strategy enumeration via exhaustive approaches [22], (2) attempt to generate an entire strategy tree [27], and (3) do not address environment measurement noise [27]. The contribution of this paper is a novel approach based on symbolic execution [19], weighted model counting [10], [7], [36], and numeric optimization [42] for the online automatic synthesis of attacks that leak maximum amount of private information, and directly addresses the above issues by (1) symbolically representing program behaviors, (2) generating strategy steps dynamically, and (3) using Bayesian inference to model adversary knowledge in a noisy environment.

```
Client:
  send l;
  receive r;

Server:
  private h = 97014599;

  private f(h, l):
    if (h ≤ l)   log.write("bound error");
    else process(l);
    return;

  while(true):
    receive l;
    f(h, l);
    send 0;
```

Figure 1: Example Client-Server application which contains a side channel.

```
min = 0;  max = 2^32;
while (min ≠ max)
    l = ⌈(min+max)/2⌉
    t = time { send l;  receive r; }
    if (t = 2 ms)  min = l;
    else max = l;
h = min;
```

Figure 2: Example side-channel attack to reveal the secret value of h stored on the server in Figure 1.

Motivating example. Consider the pseudo-code in Figure 1. A client sends a *low security* input l and gets back a response, r. On the other end, a server runs forever waiting to receive l and calls a private function $f(h, l)$, where h is initialized as a private *high security* integer variable representing a secret resource bound, and then responds with 0. The function f compares h and l, writing to an error log if l is too large.

Suppose a malicious adversary $\mathcal{A}$ wants to know the server's secret resource bound h. $\mathcal{A}$ reasons that sending an input which causes the server to write an error to the log should cause a longer round-trip time delay between send and receive than an input which does not cause this error. Now, imagine that the adversary is attacking an idealized system in which this time difference will always be the same, say, for the sake of the example, 2 ms when nothing is written to the log and 4 ms when there is a write to the log. This timing difference gives the adversary a side channel in time which can be exploited to extract the value of h. $\mathcal{A}$ can then try different values of l and decide if the value of h is larger than l or not based on the elapsed time. This enables a binary search on h. In Figure 2 we see pseudo-code for such an attack, assuming h is a 32-bit unsigned integer. This is an example of an *adaptive attack*, in which $\mathcal{A}$ makes choices of l based on prior observations. In this paper, we present techniques for *automatically* synthesizing such attacks in the presence of system noise.

Existing works on automated adaptive attack synthesis assume idealized conditions [22], [27]. However, due to noise in the server and the network, an attack which works for the idealized system is not applicable in practice. The observable elapsed time for each path of f is not a discrete constant value, but follows some probability distribution, thereby obscuring the distinguishability of program paths.

In our example, suppose that, from the client side, the timing measurements from each branch follow distributions approximately centered around 2 ms and 4 ms as in Figure 3. If $\mathcal{A}$ sends $l = 100$, observing a 1 ms duration almost certainly implies that $h > 100$ and observing a 5 ms duration almost certainly implies that $h \geq 100$. But, if $\mathcal{A}$ observes a 2.8 ms duration, it appears to be a toss-up—it could be that either $h > 100$ or $h \leq 100$ with nearly equal likelihood.

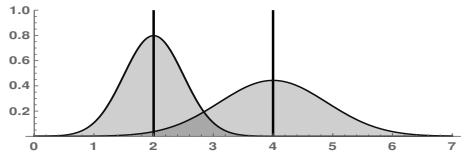


Figure 3: Distributions of timing measurements corresponding to the two branches of the server's function f from Figure 1: $h \leq l$ (left) and $h > l$ (right).

We present an approach by which an adversary automatically synthesizes inputs to extract information about secret program values despite noise in the system observables. At a high level, we perform offline profiling of a *shadow system* under our control which mimics the real system, in order to estimate the observable probability distributions. Armed with these distributions and some initial belief about the distribution of the secret, we iteratively synthesize a system input l^* which yields the largest expected information gain by solving an information-theoretic objective function maximization problem. In each attack step, the synthesized adversary 1) invokes the system with l^*, 2) makes an observation of the *real* system, 3) makes a Bayesian update on its prior belief about the secret value, and repeats the process for the next attack step.

2. Overview

In this section, we give an overview of our model for the adversary-system interaction, make explicit the program parameters over which we conduct our analysis, provide the high-level steps of dynamic attack generation, and give a discussion of relevant information theory concepts.

2.1. System Model

We use a model in which an adversary $\mathcal{A}$ interacts with a system S that is decomposed into a program P and a noise function $\mathcal{N}$, illustrated in Figure 4. The program and runtime environment form a probabilistic interactive system acting as an information channel, modeled as the probability of a noisy observation o given the inputs h and l: $p(O = o|H = h, L = l)$. An attacker who wants to learn the secret input h is interested in "reverse engineering" that probabilistic relationship. The adversary wants to compute the probability of a secret input value h given knowledge of his public input l and the side-channel observation o: $p(H = h|O = o, L = l)$. We use Bayesian inference to formalize this process of reverse engineering. By solving a numeric entropy optimization problem at each attack step, an

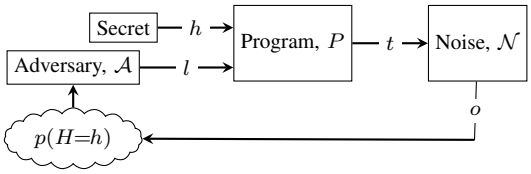


Figure 4: Model of Adversary, program, inputs, and observations as a probabilistic system.

attacker selects optimal input l^* with the goal that $p(H = h)$ converges to a distribution that assigns high probability to the actual initially unknown value of h. Below we define the components of our model.

Program under attack. We assume that P is a deterministic program, which takes exactly two inputs h and l, and we write $P(h, l)$ to indicate the invocation of P on its inputs. The ability of our model to handle programs with more than two inputs is explained in the definitions of h and l. We explain how our model can be relaxed to handle non-deterministic programs in Section 4.7.

High security inputs. By $\mathbb{H}$ we denote the set of possible *high security values*, with h being a specific member of $\mathbb{H}$, and H being a random variable which ranges over $\mathbb{H}$. These secret inputs on the real system are not directly accessible to the adversary $\mathcal{A}$ whose goal is to learn as much as possible about the value of $h \in \mathbb{H}$. We assume that the choice of h does not change over repeated invocations of P.

Low security inputs. By $\mathbb{L}$ we denote the set of *low security values*, with l being a specific member of $\mathbb{L}$, and L the corresponding random variable. These inputs are under the control of the adversary $\mathcal{A}$ who chooses a value for l and invokes P with this input. $\mathcal{A}$ may choose different values of l over repeated invocations of P.

Observations. The adversary is able to make side-channel observations, like time measurements, when the program is run. We denote the set of possible observations by $\mathbb{O}$. We assume that the set $\mathbb{O}$ is continuous for the purpose of modeling timing side channels, with O a random variable.

Program traces. A trace t is a characterization of a single program execution. We suppose that a run of P according to a trace t is manifested, from $\mathcal{A}$'s point of view, as a side-channel observation o which may be distorted by noise.

The adversary. The adversary $\mathcal{A}$ has some current belief $p(H = h)$ about the value of the secret, chooses an input l to provide to the system, and makes a side-channel observation, o, of the system for that input. $\mathcal{A}$ then makes a Bayesian update on $p(H = h)$ based on o and repeats the process. The contribution of this paper is the synthesis of the optimal inputs, l^*, which cause the system to leak the most information about h.

2.2. Outline of Attack Synthesis

Our attack synthesis method is split into two phases. The reader can refer to Figure 5 for this discussion.

Phase 1: Offline static analysis and profiling. The main goal of the offline phase is to estimate a probabilistic relationship between program traces and side-channel observations. Informally, a program trace is a sequence of program states, including control flow choices at each branch condition. We group program traces into trace classes based on indistinguishability via observation. We summarize the main points of this phase below, and provide the detailed discussions in Section 3.

1) We use symbolic execution to compute path constraints (PCs) on the secret and public inputs for the program source code. Path constraints are logical formulas over inputs that characterize an initial partition estimate for program traces. Each PC, ϕ_i, is associated with a trace class T_i, (Section 3.2).
2) For each PC, ϕ_i, we generate a witness $w_i = (h_i, l_i)$. Each w_i is assumed to be a characteristic representative of all concrete secret and public inputs that cause P to execute any of the traces in a trace class T_i.
3) For each w_i, we repeatedly run the system with w_i as input and record observation samples. From the samples, we estimate the conditional probability of observation given trace class, denoted $P(O|T = T_i)$ (Section 3.3).
4) PCs may generate too fine a partition of program trace classes. That is, there may be two trace classes T_i and T_j where $P(O|T = T_i)$ and $P(O|T = T_j)$ coincide to such a degree that the traces are effectively indistinguishable by measuring O. Thus, we merge PC's and corresponding distributions which are too similar according to a metric known as the Hellinger distance (Section 3.4).

Phase 2: Online dynamic attack synthesis. The second phase mounts an adaptive attack against the system, making use of the estimated system profile from the offline phase, assuming that the adversary has some initial belief about the distribution of the secret. We summarize the main approach for this phase below, with detailed discussions in Section 4.

1) We use the current belief about the secret along with path constraints to compute, for each ϕ_i, a model-counting function which is symbolic over the public inputs, denoted $f_i(l)$(Section 4.2).
2) We use the model-counting functions to compute trace class probabilities as symbolic functions over low security inputs. We then apply the mutual information formula from information theory to get an information leakage objective function, which is symbolic over the public inputs (Section 4.3).
3) We use numeric optimization to compute the public input l^* that maximizes the symbolic information leakage objective function (Section 4.4).
4) We provide the leakage-maximizing input, l^*, to the system and record the observation.
5) We use Bayesian updates to refresh the current belief about the secret using the observation and the noise

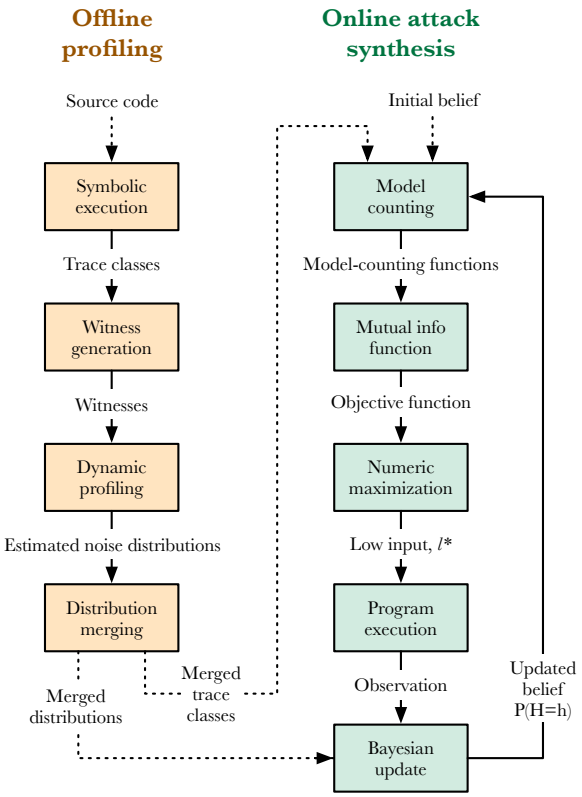


Figure 5: Overview of our attack synthesis approach.

profile that was estimated during the offline phase (Section 4.5). We repeat this process starting from Step 1.

2.3. Measuring Uncertainty

In order to maximize information leakage we must have a way to quantify it. Here we give a brief background on information-theoretic concepts we use in our approach. Intuitively, before invoking P, $\mathcal{A}$ has some *initial uncertainty* about the value of h, while after observing o, some amount of *information is leaked*, thereby reducing $\mathcal{A}$'s *remaining uncertainty* about h. The field of quantitative information flow (QIF) [38] formalizes this intuition by casting the problem in the language of information theory using *Shannon's information entropy* which can be considered a measurement of uncertainty [37]. We briefly give three relevant information entropy measures [13]. Given a random variable X which can take values in $\mathbb{X}$ with probabilities $p(X = x)$, the *information entropy* of X, denoted $\mathcal{H}(X)$ is given by

$$\mathcal{H}(X) = \sum_{x \in \mathbb{X}} p(X = x) \log_2 \frac{1}{p(X = x)} \tag{1}$$

Given another random variable Y and a conditional probability distribution $p(X = x|Y = y)$, the *conditional entropy of X given Y* is

$$\mathcal{H}(X|Y) = \sum_{y \in \mathbb{Y}} p(Y = y) \sum_{x \in \mathbb{X}} p(X = x|Y = y) \log_2 \frac{1}{p(X = x|Y = y)} \tag{2}$$

Intuitively, $\mathcal{H}(X|Y)$ is the expected information contained in X given knowledge of Y. Given these two definitions, we would like to compute the expected information gained about X by observing Y. In our application, we target timing side channels where we model time as a continuous random variable, while the secret is a discrete value (i.e., an integer or a string). In order to measure the mutual information between a discrete random variable Y and a continuous random variable X, we use the Kullback–Leibler (KL) divergence [13].

The KL divergence, $D_{\mathrm{KL}}(p, q)$ is a statistical measure of the discrepancy between two models, $p(x)$ and $q(x)$, for a probabilistic event X over a continuous domain. It is computed via the formula:

$$D_{\mathrm{KL}}(p, q) = \int_{-\infty}^{\infty} p(x) \log \frac{p(x)}{q(x)} \, \mathrm{d}x. \tag{3}$$

Then, the *mutual information* between a discrete random variable Y and a continuous random variable X is defined as the expected information gain with expectation taken over all possible events in Y:

$$\mathcal{I}(Y; X) = \sum_{y \in Y} p(Y = y) D_{\mathrm{KL}}(p(X|Y = y), p(X)) \tag{4}$$

Intuitively, $D_{\mathrm{KL}}(p(X|Y = y), p(X))$ is a measure of information gain between the prior distribution $p(X)$ and the posterior distribution $p(X|Y = y)$.

We consider the high security input, low security input, and observable as random variables H, L, and O, where H and L are discrete and O is continuous. We interpret $p(H)$ as the adversary's initial belief about h and $\mathcal{H}(H)$ as the initial uncertainty. The conditional entropy $\mathcal{H}(H|O, L = l)$ quantifies $\mathcal{A}$'s remaining uncertainty after providing input $L = l$ and observing output O. Finally, we interpret $\mathcal{I}(H; O|L = l)$ as the amount of information leaked.

The goal of the adversary is to maximize the value of $\mathcal{I}(H; O|L = l)$ at every step, which we call the leakage for that step, measured in bits of information. Our attack synthesis technique relies on choosing an optimal input $L = l^*$ at each step which maximizes the expression given in Equation 4. At a high level, we compute a symbolic expression for $\mathcal{I}(H; O|L = l)$ using symbolic weighted model counting and then use off-the-shelf numeric maximization routines to choose l^* (detailed in Section 4).

3. Offline Profiling

The first phase of our attack synthesis relies on offline pre-computation of equivalence classes of program traces and corresponding noise distributions. We accomplish this through the use of symbolic execution, dynamic profiling using a small representative set of witness inputs, and statistical measures for merging indistinguishable distributions. The results of this offline phase are then used during the online attack synthesis phase described in Section 4.

3.1. Trace Equivalence Classes

Let $\mathbb{T}$ denote the set of execution traces of program P. A trace is a characterization of a single program execution. From adversary $\mathcal{A}$'s point of view, invoking P induces execution of a single trace t. Knowing the input l and the executed program trace t, $\mathcal{A}$ can gain information about h. However, $\mathcal{A}$ does not which program trace t was executed, but can make a side-channel observation $o \in \mathbb{O}$, which may be distorted by system noise. There are two challenges:

1) **Observation noise:** the same execution trace t may lead to different observations o_1 and o_2 in different runs of P.
2) **Observation collision:** two different traces t_1 and t_2 may lead to the same observation o in some runs of P.

We address these two challenges by defining *trace classes*, which identify observationally equivalent traces in the presence of noise. Let T be a random variable that ranges over $\mathbb{T}$ and $p(O|T)$ be the conditional probability density function on observations conditioned on which trace is executed. We define an equivalence relation $\cong$ on $\mathbb{T}$ in which $t_1 \cong t_2$ if $p(O|T = t_1) = p(O|T = t_2)$ and we say t_1 and t_2 are observationally equivalent. Let partition $\mathcal{T}$ be the set of equivalence classes of $\mathbb{T}$ defined by $\cong$. Then, we call each $T_i \in \mathcal{T}$ a *trace class*.

$\mathcal{A}$ gains information about h by knowing which trace t was executed when P is run with l. But due to noise and collisions, the best $\mathcal{A}$ can hope for with a single run of the program is to determine the likelihood that $t \in T_i$ for each trace class T_i. So, given l and o, $\mathcal{A}$ would like to know $p(T \in T_i|O = o, L = l)$. In the remainder of this section we show how $\mathcal{A}$ can compute a characterization of $\mathcal{T}$ and efficiently estimate $p(O|T \in T_i)$. We explain in Section 4 how $p(O|T \in T_i)$ is used during the online attack phase to compute $p(T \in T_i|O = o, L = l)$.

3.2. Trace Class Discovery via Symbolic Execution

We now describe how symbolic execution can be used as a first approximation of trace classes. First, we briefly describe symbolic execution and then explain how symbolic execution's path constraints are associated with trace classes.

Symbolic execution [19] is a static analysis technique by which a program is executed on *symbolic* (as opposed to concrete) input values which represent all possible concrete values. Symbolically executing a program yields a set of *path constraints* $\Phi = \{\phi_1, \phi_2, \ldots, \phi_n\}$. Each ϕ_i is a conjunction of constraints on the symbolic inputs that characterize all concrete inputs that would cause a path to be followed. All the ϕ_i's are disjoint. Whenever symbolic execution hits a branch condition c, both branches are explored and the constraint is updated: ϕ becomes $\phi \wedge c$ in the *true* branch and $\phi \wedge \neg c$ in the *false* branch. Path constraint satisfiability is checked using constraint solvers such as Z3 [17]. If a path constraint is found to be unsatisfiable, that path is no longer analyzed. For a satisfiable path constraint, the solver can return a model (concrete input) that will cause that path to be executed. To deal with loops and recursion, a bound is typically enforced on exploration depth.

We use the path constraints generated by symbolic execution as an initial characterization of trace classes, where we consider any secret inputs as a vector of *symbolic high security variables* h and remaining inputs as a vector of *symbolic low security variables* l. Then symbolic execution results in a set of path constraints over the domains $\mathbb{H}$ and $\mathbb{L}$, which we write as $\Phi = \{\phi_1(h, l), \phi_2(h, l), \ldots, \phi_n(h, l)\}$. In general, we find it useful to think of each $\phi_i(h, l)$ as a logical formula which returns true or false, but sometimes it is convenient to think of $\phi_i(h, l)$ as a characteristic function that returns 1 when the path constraint is satisfied by h and l and 0 otherwise. A *witness* $w_i = (h_i, l_i)$ for a given ϕ_i is a concrete choice of $h = h_i$ and $l = l_i$ so that $\phi_i(h_i, l_i)$ evaluates to true, and we write $w_i \vDash \phi_i(h, l)$.

We assume that inputs that satisfy the same path condition ϕ_i induce traces that are observationally equivalent, and hence belong to the same trace class T_i. I.e., for inputs (h, l) and (h', l'), if $(h, l) \vDash \phi_i$ and $(h', l') \vDash \phi_i$, then running $P(h, l)$ and $P(h', l')$ results in traces t_1 and t_2 that reside in the same trace class T_i. Thus, our characterization of trace classes is defined by the set of path constraints Φ, where each path constraint ϕ_i is associated with a trace class T_i. Recalling the example from Figure 1, the trace class characterization based on path constraints is $\{h \leq l, h > l\}$.

In the following two subsections we discuss how we address the issues of observation noise and observation collision for the trace classes induced by path constraints.

3.3. Estimating Observation Noise

In order to estimate the trace-observation noise, for each ϕ_i we find a witness that satisfies ϕ_i and profile the program running in the environment with that input. Witness generation is a common practice in symbolic execution in order to provide concrete inputs that demonstrate a particular program behavior. Many satisfiability solvers support witness generation. Our method relies on the assumption that any witness $w_i = (h_i, l_i)$ for a path condition ϕ_i has a side-channel observation characteristic that is representative of all other inputs that satisfy ϕ_i. That is,

$$(h, l) \vDash \phi_i \Rightarrow p(O|H = h, L = l) = p(O|H = h_i, L = l_i)$$

Trace-class sampling. Assuming that P is deterministic, every (h, l) pair results in exactly one trace, and the side-channel observation relationship for every (h, l) pair is characterized by a witness w_i for the path constraint ϕ_i that corresponds to trace class T_i. Hence, we assume that $p(O|T \in T_i) = p(O|H = h_i, L = l_i)$. Thus, in order to estimate the effect of system noise on a trace class, we repeatedly collect observation samples using w_i as a representative input (Procedure 1).

Distribution estimation. Given a sample set of observations, $\mathcal{A}$ can estimate the probability of an as-yet-unseen observation with well-known density function interpolation methods using *smooth kernel density estimation*. Suppose $\{o_1, o_2, \ldots, o_n\}$ is a set of independent and identically

Procedure 1 System S, Path Constraints Φ, # of Samples m

```
1: procedure PROFILE(S, Φ, n)
2:     for each φ_i ∈ Φ
3:         (h_i, l_i) ← GENERATEWITNESS(φ_i)
4:         for j from 1 to m
5:             SAMPLE(i, j) ← S(h, l)
```

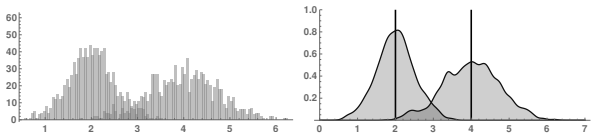


Figure 6: Histogram of 1000 timing samples for trace classes T_1 and T_2 (left) and the smooth kernel density estimates for $p(O|T \in T_i)$ (right).

distributed samples drawn from the unknown distribution $p(O|T \in T_i)$ for a specific trace class T_i. We want to estimate the value of $p(O = o|T \in T_i)$ for an unseen sample o. The *kernel density estimator* $\hat{p}(O|T \in T_i)$ is

$$\hat{p}(O = o|T \in T_i) = \frac{1}{nh} \sum_{i=1}^{n} K\Big(\frac{o - o_i}{h}\Big) \tag{5}$$

where K is a smoothing kernel and h is a smoothing parameter called the bandwidth [34], [26]. We use a Gaussian distribution for the smoothing kernel K and we use a bandwidth that is inversely proportional to the sample size. Using Procedure 1 and Equation 5, we have an estimate for the effect of the noise on each trace class T_i.

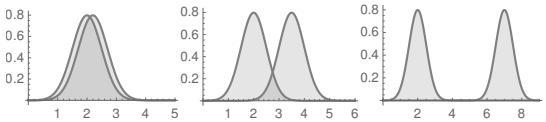
For the example from the introduction, there are two trace classes T_1 and T_2 corresponding to path conditions $\phi_1(h, l) = h \leq l$ and $\phi_2(h, l) = h > l$. We use a constraint solver to find witnesses $w_1 = (4, 10)$ and $w_2 = (9, 3)$ so that $w_1 \models \phi_1$ and $w_1 \models \phi_2$. Then, running the system with w_1 and w_2 for 1000 timing samples results in the histograms and the the corresponding smooth kernel estimate distributions shown in Figure 6.

3.4. Trace Class Merging Heuristic

It is possible that the set of path constraints Φ that characterize $\mathcal{T}$ are an over-refinement of the actual trace classes of P. It may be that two trace classes are effectively indiscernible via observation due to system noise. For this reason, we employ a heuristic which combines path constraints when their corresponding estimated probability distributions are too similar. We measure the similarity of two distributions using the Hellinger distance $d_H(p, q)$ between density functions p and q given by

$$d_H(p, q) = \sqrt{\frac{1}{2} \int_{-\infty}^{\infty} \Big(\sqrt{p(x)} - \sqrt{q(x)}\Big)^2 \, dx}$$

The Hellinger distance is such that $0 \leq d_H(p_1, p_2) \leq 1$. Intuitively, $d_H(p_1, p_2)$ measures distance by the amount of "overlap" between p and q: $d_H(p, q)$ is 0 if there is perfect overlap and 1 if the two distributions are completely disjoint



(a) $d_H = 0.068$ (b) $d_H = 0.491$ (c) $d_H = 0.978$

Figure 7: Three pairs of probability distributions and their Hellinger distances. With a threshold of $\tau = 0.1$, the distributions in (a) are indistinguishable while the distributions in (b) and (c) are distinguishable.

(see Figure 7). We merge path conditions ϕ_i and ϕ_j if $d_H(\hat{p}(O|T \in T_i), \hat{p}(O|T \in T_j)) \leq \tau$ for a threshold τ.

4. Online Attack Synthesis

In this section we describe how to automatically synthesize an attack for the adversary.

4.1. Adversary Strategy

Recall the system model described in Section 1 and Figure 4 in which the adversary $\mathcal{A}$ repeatedly interacts with the system, trying to discover information about h. We define the strategy for the adversary $\mathcal{A}$ in Procedure 2 and summarize the steps taken in our model of $\mathcal{A}$.

Procedure 2 System S, Current belief $p(H)$

```
1: procedure ATTACK(S, p(H))
2:     l ← CHOOSELOWINPUT(p(H))
3:     o ← S(h, l)
4:     p(H) ← p(H|O = o, L = l)
5:     ATTACK(S, p(H))
```

1) **Input choice.** $\mathcal{A}$ executes an attack step by choosing an input l^* that will maximize the expected information leakage. The method for choosing l^* is the core of our technique, involving the numeric maximization of an objective leakage function which is symbolic over l, computed via symbolic weighted model counting. Sections 4.2, 4.3, and 4.4 detail this step.
2) **Program invocation.** The adversary submits input l so that $P(h, l)$ is invoked.
3) **Record observation.** As a result of invoking the system, $\mathcal{A}$ records a side-channel observation o.
4) **Belief update**. $\mathcal{A}$ uses the current belief $p(H)$ and trace class probabilities to make a Bayesian update on the current belief about h given input l^* and observation o. That is, $\mathcal{A}$ sets the current belief $p(H = h) \leftarrow p(H = h|O = o, L = l)$ (Section 4.5).
5) **Repeat.** Run the attack on S with the updated belief.

The step $l \leftarrow$ CHOOSELOWINPUT$(p(H))$ is an entropy maximization procedure consisting of several components. We use weighted model counting to generate expressions that quantify the probabilities of trace classes given an input l, which is kept symbolic, and the current belief about the

secret, $p(H)$. These probability expressions can be used along with the estimated noise distributions to express the mutual information $\mathcal{I}(O; H|L = l)$. Alternatively, one may consider the mutual information with regard to trace classes, $\mathcal{I}(T; H|L = l)$. In either case, we use numeric routines to find the input which maximizes $\mathcal{I}$. Our experiments demonstrate that maximizing $\mathcal{I}(T; H|L = l)$ is considerably faster and still generates efficient attacks.

4.2. Trace Class Probabilities via Symbolic Weighted Model Counting

When $\mathcal{A}$ selects an input l, the system runs $P(h, l)$ resulting in a trace which cannot be directly known to $\mathcal{A}$ using side-channel observations. However, $\mathcal{A}$ can compute the probability that the trace belongs to trace class T_i, i.e., $p(T \in T_i|L = l)$, relative to his current belief about the distribution of the secret $p(H)$. If P is deterministic, for a concrete choice of l the probability of a particular trace t relative to the current belief about h is simply the probability of the particular value of h that induces t; $p(T = t_i|L = l) = p(H = h)$. Furthermore, recall from Section 3.2 that each trace class T_i is associated with a path constraint on the inputs $\phi_i(h, l)$, and think of $\phi_i(h, l)$ as a characteristic function that returns 1 if running $P(h, l)$ results in a trace t from trace class T_i and returns 0 otherwise. Then, we have the following equivalence between trace class probabilities with respect to a choice of l, the current belief $p(H)$ and the path constraints Φ:

$$p(T \in T_i|L{=}l) = \sum_{t \in T_i} p(T{=}t_i|L{=}l) = \sum_{h \in \mathbb{H}} p(H{=}h)\phi_i(h, l) \quad (6)$$

The resulting expression on the right hand side is an instance of a *weighted model counting* problem. Unweighted model counting is the problem of counting the number of models of a formula. If one assigns a weight to each model, one may compute the sum of the weights of all models. Thus, the unweighted model counting problem is a special instance of the weighted model counting problem obtained by setting all weights equal to 1. In our case, we are interested in counting models of ϕ_i where the weight of each model (h, l) is given by $p(H = h)$.

Clearly, one may compute Equation (6) by summing over $\mathbb{H}$ but this would be inefficient. For instance, $\mathbb{H}$ may be large and Equation (6) will need to be recomputed every time $p(H)$ is updated. In addition, in the numeric maximization procedure, which we describe later, we would like to be able to evaluate Equation (6) for many different values of l. Hence, we seek an efficient way to compute the weighted model count. This is accomplished by using symbolic weighted model counting. We first give an example.

Recall the path constraints from the example in Figure 1 where $\phi_1(h, l) = h \leq l$ and $\phi_2(h, l) = h > l$ corresponding to two trace classes T_1 and T_2. Additionally, suppose that $\mathcal{A}$'s current belief is that $1 \leq h \leq 24$ and that larger values of h are more likely to occur. $\mathcal{A}$ models his initial belief as a probability distribution

$$p(H = h) = \begin{cases} \frac{h}{300} & 1 \leq h \leq 24 \\ 0 & \text{otherwise} \end{cases}$$

We can compute symbolic formulas for $p(T \in T_i|L = l)$ as functions of his choice of input l:

$$p(T \in T_1|L = l) \quad = \quad \begin{cases} 0 & l < 1 \\ \frac{l^2+l}{600} & 1 \leq l \leq 24 \\ 1 & l > 24 \end{cases}$$

$$p(T \in T_2|L = l) \quad = \quad \begin{cases} 1 & l < 1 \\ 1 - \frac{l^2+l}{600} & 1 \leq l \leq 24 \\ 0 & l > 24 \end{cases}$$

When $\mathcal{A}$ wants to compute the probability that the program will execute a trace from any trace class for a particular choice of l, he may simply evaluate these functions for that choice of l. For example, if $\mathcal{A}$ inputs $l = 10$ he expects to observe a trace from T_1 with probability $11/60$ and a trace from T_2 with probability $49/60$. Using trace class probability functions that depend on l, $\mathcal{A}$ can efficiently compare the likelihood that the program executes a trace from any trace class depending on the choice of l. These symbolic probability functions are used in the next section to generate a symbolic information gain function which is used as the objective of numeric maximization.

We compute a symbolic function for $p(T \in T_i|L = l)$ by using a model-counting constraint solver. Our implementation targets functions whose path constraints can be represented as boolean combinations of linear integer arithmetic constraints and we use the model counting tool Barvinok [7], [40] for this purpose. For programs that operate on strings, we interpret strings as arrays of integers that are bounded to be valid byte values of ASCII characters.

Barvinok performs weighted model counting by representing a linear integer arithmetic constraint ϕ on variables $X = \{x_1, \ldots, x_n\}$ with weight function $W(X)$ as a symbolic polytope $Q \subseteq \mathbb{R}^n$. Let $Y \subseteq X$ be a set of parameterization variables and Y' be the remaining free variables of X. Barvinok's polynomial-time algorithm generates a (multivariate) piecewise polynomial F such that $F(Y)$ evaluates to the weighted count of the assignments of integer values to Y' that lie in the interior of Q. We are interested in computing the probability of a trace class given a choice of l and the current belief about the high security inputs. Thus, we let $Y = \mathbb{L}$, $Y' = \mathbb{H}$, and W be $p(H = h)$.

4.3. Leakage Objective Function

Here we describe how to generate an objective function that quantifies the amount of information that $\mathcal{A}$ can gain by observing o after sending input l. The major point of this section is to show that it is possible to express the information leakage as a symbolic function using weighted model-counting functions and the estimated noise distributions.

Mutual Information Between Secret and Observation. For a given choice of l we can quantify $\mathcal{I}(H; O|L = l)$ by directly applying the definition of mutual information, Equation (4).

$$\sum_{h\in\mathbb{H}} p(H=h) D_{\mathrm{KL}}(p(O=o|H=h, L=l), p(O=o|L=l))$$

Because the path constraints Φ are disjoint, and cover all paths of the program, for a particular l, Φ determines a partition on h. Thus, we can rewrite $\mathcal{I}(H;O|L=l)$ by summing over path conditions:

$$\sum_{i=1}^{n}\sum_{h\in\mathbb{H}} p(H{=}h)\phi_i(h,l) D_{\mathrm{KL}}(p(O{=}o|H{=}h, L{=}l), p(O{=}o|L{=}l))$$

Since $\phi_i(h,l)=0$ unless input (h,l) induces trace class T_i and the observation probability is conditioned on the trace class, we rewrite the expression in terms of trace classes:

$$\sum_{i=1}^{n} D_{\mathrm{KL}}(p(O{=}o|T{\in}T_i), p(O{=}o|L{=}l)) \sum_{h\in\mathbb{H}} p(H{=}h)\phi_i(h,l)$$

Observe that the sum over $\mathbb{H}$ is exactly what is computed via symbolic weighted model counting in Equation (6); thus, $\mathcal{I}(H;O|L=l)$ can be expressed as

$$\sum_{i=1}^{n} D_{\mathrm{KL}}(p(O{=}o|T{\in}T_i), p(O{=}o|L{=}l)) p(T{\in}T_i|L{=}l) \quad (7)$$

where D_{KL} is computed with Equation (3) and the probability of the observation conditioned on the low input choice is a straightforward conditional probability computation

$$p(O=o|L=l) = \sum_{i=1}^{n} p(O=o|T\in T_i)\cdot p(T\in T_i|L=l) \quad (8)$$

Thus, $\mathcal{I}(H;O|L=l)$ can be computed via Equations (7) and (8) using only $p(O=o|T\in T_i)$ (estimated via sampling and smooth kernel interpolation) and $p(T\in T_i|L=l)$ (computed via symbolic weighted model counting).

Mutual information between secret and trace classes. The approach to mutual information quantification based on Equation (7) relies on integrating over the domain $\mathbb{O}$, an expensive computation. Alternatively, as a heuristic, one may compute the mutual information between the secret and the trace classes given the low input.

$$\mathcal{I}(H;T|L=l) = \mathcal{H}(T|L=l) - \mathcal{H}(T|H, L=l)$$

Note that if P is deterministic, T is completely determined by H and L and so $\mathcal{H}(T|H, L=l)=0$. Thus,

$$\mathcal{I}(H;T|L{=}l) = \sum_{i=1}^{n} p(T{\in}T_i|L{=}l)\log\frac{1}{p(T{\in}T_i|L{=}l)} \quad (9)$$

While this is not guaranteed to give the optimal attack, it can be computed much more efficiently than Equation (7). Equation (9) can be quickly evaluated for many choices of l using the symbolic functions for $p(T\in T_i|L=l)$ because they are computed by Barvinok as symbolic weighted model-counting functions. In Section 5 we will see that quantifying information leakage in this way allows for more efficient generation of attack inputs.

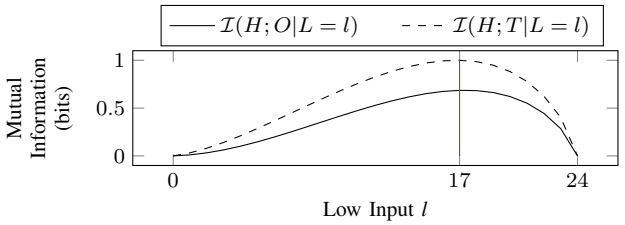


Figure 8: Mutual information between secret H and observation O or trace classes T as a function of low input $L=l$ for the example from Figure 1.

Procedure 3 Noise Entropy Aware Input Choice.
Current belief $p(H)$, path constraints Φ, noise $p(O|L)$

```
1: procedure ChooseLowInput(p(H), Φ, p(O = o|L = l))
2:     for each φ_i ∈ Φ
3:         p(T ∈ T_i|L = l) ← Barvinok(p(H), Φ)
4:     f(l) ← I(H;O|L = l) via Eq. 7
5:     // or f(l) ← I(H;T|L = l) via Eq. 9
6:     l* ← NMaximize(f(l))
7:     return l*
```

Recall the example from Figure 1. After using symbolic execution to determine path constraints associated with trace classes, estimating the noise, and performing symbolic weighted model counting, we can compute both $\mathcal{I}(H;O|L=l)$ and $\mathcal{I}(H;T|L=l)$, plotted in Figure 8. Observe that the expected information leakage computed from Equation (7) is strictly less than the one computed using only trace classes based on Equation (9). So, the trace class information gain bounds the actual information gain. However, both maxima occur at the same value of $l^*=17$. Hence, both methods agree on the optimal choice of input. While we do not claim that the two maxima coincide in general, our experiments support that maximizing $\mathcal{I}(H;T|L=l)$ rather than $\mathcal{I}(H;O|L=l)$ is significantly faster per attack step, but may result in slight longer attacks.

4.4. Input Choice via Numeric Optimization

Now, using Equation (7) or Equation (9), $\mathcal{A}$ can apply numeric optimization procedures to choose an input l^* that maximizes the information gain. $\mathcal{A}$ can compute

$$l^* = \arg\max_l \mathcal{I}(H;T|L{=}l) \text{ or } l^* = \arg\max_l \mathcal{I}(H;O|L{=}l)$$

These are non-linear combinatorial optimization problems. We can use black-box objective function maximization routines to find the values of l that maximize the leakage. These routines are typically stochastic, and guaranteed to produce a local optimum, but not necessarily a global one. Regardless of how l^* is chosen, $\mathcal{A}$ can still efficiently and precisely update $p(H)$ based on l^* and the resulting system side-channel observation which we detail in Section 4.5. The two methods are given as Procedure 3 using Eq. 7 or Eq. 9.

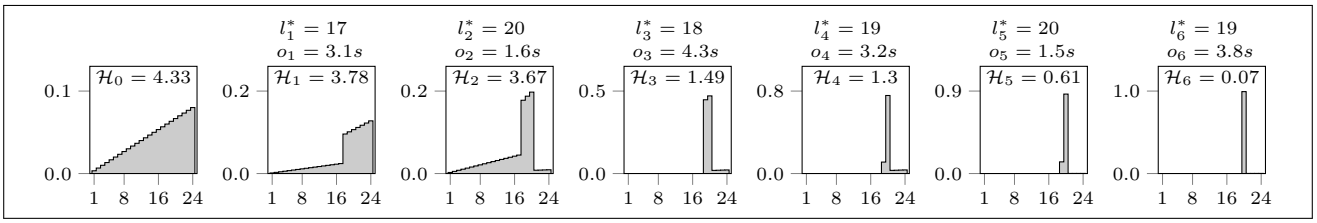


Figure 9: A sequence of attack steps indicating $\mathcal{A}$'s changing belief about the secret $p(H = h)$ after making input l_i^* and observing o_i at the i^{th} step. The current uncertainty, $\mathcal{H}_i$ (bits), is indicated.

4.5. Belief Update for Secret Distribution

After providing input l^* and making side-channel observation o, $\mathcal{A}$ will need to refresh his current belief about the secret $p(H = h)$ by performing the update $p(H=h) \leftarrow p(H=h|O=o, L=l)$. Although this is a straightforward Bayesian update, we provide the formula in order to illustrate how $p(T \in T_i|L=l)$, computed via weighted model counting, and $p(O=o|T \in T_i)$, estimated via Procedure 1, are involved in the update. By applying Bayes' rule and the definitions of conditional probability and summing over trace classes we have that

$$p(H=h|O=o, L=l^*) = \sum_{i=1}^{n} p(H=h)\phi_i(h,l)\frac{p(O=o|T \in T_i)}{p(O=o|L=l^*)} \tag{10}$$

where $p(O=o|L=l)$ is computed from $p(T \in T_i|L=l)$ and $p(O=o|L=l)$ using Equation (8). Equation (10) allows $\mathcal{A}$ to easily update $p(H)$ by simply plugging the value of l^* that was used as input and the resulting observation o.

4.6. Example

Recall the example from Figure 1 and suppose $\mathcal{A}$ has an initial belief about the secret: $p(H = h) = h/300$ if $1 \leq h \leq 24$ and 0 otherwise (Figure 9, far left). There are two path constraints $\phi_1(h, l) = h \leq l$ and $\phi_2(h, l) = h > l$, which result in trace class probability functions $p(T \in T_i|L = l)$ as in Section 4.2.

As we saw in Section 4.3, the optimal input of the first step of an attack is $l_1^* = 17$. In Figure 9 we show 6 steps of an attack in which the secret is $h = 20$ and we start by making input $l^* = 17$. Suppose that when making the input, $\mathcal{A}$ makes a timing measurement and observes $o_1 = 3.1$ms Then, $\mathcal{A}$ updates his belief about h as shown in the second step of Figure 9. Recalling the noise estimates in Figure 6, we see that for a timing measurement of 3.1ms, it is more likely that a trace from trace class T_2 corresponding to $\phi_2(h, l) = h > 17$ occurred, than a trace from trace class T_1 corresponding to $\phi_1(h, l) = h \leq 17$. Consequently, after performing the Bayesian update, the probability mass to the right of $h = 17$ increases and the probability mass to the left decreases proportionally according to Equation (10).

$\mathcal{A}$ continues choosing inputs and making observations for 6 steps. We see that at different steps of the attack, the optimal input is repeated from a previous step: $l_4 = l_6 = 19$, for instance; this technique automatically performs a form of repeated sampling in order to improve confidence. After 6 steps the probability mass of $p(H = h)$ is concentrated on $h = 20$ and the corresponding uncertainty is $\mathcal{H}_6 = 0.07$ bits, and so $\mathcal{A}$ may reasonably conclude that $h = 20$ with high probability. Finally, we note that CHOOSELOWINPUT gives identical attacks if provided the same observation sequences from Figure 9 using either Eq. 7 and Eq. 9.

4.7. Handling Non-deterministic Programs

Although our model assumes a deterministic program, we are able to handle non-deterministic programs in some cases. Programs may contain explicit randomization through the use of random number generators. Indeed, there are attempts to mitigate side-channel leakage by introducing randomization into the program [25], [14]. If a program has a random component that does not affect the branch conditions on h and l, then the randomization is essentially decoupled from the computation on h and l. Thus, running symbolic execution on such a program will yield path constraints on h and l that can be used to characterize trace classes. Then, dynamic profiling with witnesses for each path constraint will capture the effect of non-deterministic choices in the code on the observation. Hence, the deterministic component of the program that computes over h and l and the random component can be factored into the static analysis of symbolic execution and the dynamic analysis of profiling, respectively. We demonstrate in our experiments that we are indeed able to use our analysis on programs which contain explicit non-determinism in the code.

4.8. Detecting Non-vulnerability

Some programs are not vulnerable to adaptive side-channel attacks with respect to a chosen observable. We demonstrate our ability to report non-vulnerability in our experimental evaluation. Our approach is able to report non-vulnerability in two ways:

1) If all trace classes initially defined by the path constraints are determined to be observationally indistinguishable using the Hellinger distance merging metric, we conclude that there is only one trace class, and therefore, different inputs cannot leak information about the secret.

2) If the first step of attack generation determines that there is no input l^* which results in positive information gain, then again we conclude that there is no attack.

Note that trace-class constraints are generated with respect to a semantic model of a program which is an abstraction of real system behavior. In our implementation, our semantic model is based on path constraints generated from branch instructions in Java byte-code, and so our abstraction does not capture lower level details like thread scheduling or caching. The power of our technique is relative to the abstraction level used in constraint generation and the cost model. Hence, our reports of non-vulnerability are made with respect to the chosen level of semantic abstraction.

5. Implementation and Experiments

We implemented our technique according to the high-level diagram shown in Figure 5 and described in Section 2.2. We ran our attack synthesis system on client-server programs created by DARPA (Defense Advanced Research Projects Agency) for the ongoing STAC (Space/-Time Analysis for Cybersecurity) [15] research program. We evaluated the effectiveness of our approach on several programs taken from their publicly available repository [33]. These programs were crafted to test and evaluate techniques for detecting side-channel vulnerabilities in networked Java applications.

5.1. Experimental Setup

Reference platform. For all experiments, the system under test (SUT) was run on the official DARPA STAC Reference Platform [33], which specifies an Intel NUC 5i5RYH computer with an Intel Core i5-5250 CPU running at 1.60 GHz, 16 GB of DDR3 RAM, and its built-in Intel 1000 Mbps Ethernet interface. The reference operating system is CentOS 7, release 7.1.1503, with a Linux 3.10.0-229 64-bit kernel. We used the OpenJDK 64-bit Java VM, build 1.8.0_121.

Trace class extraction. We infer trace classes by symbolically executing the SUT source code. We used Symbolic Path Finder (SPF), an extension to NASA's Java Path Finder v8.51 using the Z3 SMT-solver [17], version 4.4.1.

Automated profiling. This involves two components: a client-side PROFILER and a server-side APPSERVER. The server component is a wrapper for DARPA STAC canonical challenge programs. Although their interfaces were slightly modified to achieve a homogeneous input/output format, the core implementation of each challenge program was left unmodified. The client-side PROFILER is a Python script that invokes the server while taking measurements. Given a list of trace-class witnesses, and the number of samples per witness, the PROFILER configures the server and repeatedly interacts with it over the network, carefully timing each interaction. For our experiments, each trace class witness is used to sample the system 1000 times.

Model counting. Symbolic weighted model-counting functions are computed by sending the path constraints and $p(H{=}h)$ to the Barvinok model counter [40], version v0.39.

Numeric and symbolic computing. Many computationally intensive numeric and symbolic operations are handled by Mathematica, including smooth kernel probability density estimation from timing samples, symbolic manipulation of model-counting and information-theoretic functions, numeric integration, objective function maximization, and Bayesian updating. All experiments were run using Wolfram Mathematica v11 [41] on a Linux 64-bit system.

Entropy maximization. We used Mathematica's NMAXIMIZE function [42] in *Differential Evolution* mode, which uses a fast and robust genetic algorithm [16], [43].

5.2. Experimental Evaluation

Our experiments show that our approach is able to dynamically synthesize attack input sequences for vulnerable programs and to report non-vulnerability for programs for which an attack is not feasible. In Table 1, we label the DARPA benchmark programs with a number that is consistent with the numbering from their repository along with '(v)' or '(nv)' to indicate vulnerable or non-vulnerable programs according to DARPA's classification. $|\Phi|$ denotes the number of path conditions, $|\mathcal{T}|$ denotes the number of trace classes after performing Hellinger-based merging, and $\mathrm{Dim}(h)$ indicates the size of the symbolic integer vector used to represent h. For STAC programs 1, 3, and 11, we varied the secret search domain ($2^8, 2^{16}, 2^{24}, 2^{31}$), while keeping the code the same, and so only a single offline phase was needed. For instance, row 1 of Table 1 corresponds to the offline phase of 4 different online attacks.

The STAC benchmark also contains programs that are not related to side-channel problems, so we did not analyze them. We analyzed only problems marked by DARPA as side-channel related and that fit our model. There were two side-channel problems that require a different cost observation model than ours, so we did not analyze those programs. Initial secret distributions in our experiments are uniform.

Non-vulnerable programs. In Table 1 we present results of running our attack synthesizer on two programs. We see that STAC–1(nv) is not vulnerable to an adaptive timing side-channel attack because there is only one trace class and so there are no attack steps taken. On the other hand, our tool tells us that STAC–3(nv) is not vulnerable, despite having 3 observationally distinguishable trace classes, because after 1 attack step there are no inputs which can leak any information. Thus, we agree with the DARPA non-vulnerable classification. For both programs we observe that the majority of the analysis time is spent in offline profiling.

Optimizing observation vs trace-class entropy. We applied both CHOOSELOWINPUT1, which optimizes based on observation entropy, and CHOOSELOWINPUT2, which optimizes based on trace class entropy, for two STAC programs. STAC–1(v) is similar to our running example with a branch condition $h \leq l$ in which extra computation is done.

TABLE 1: Experimental data for publicly available STAC benchmarks [33].

Benchmark		Dim(H)	$\lvert\mathbb{H}\rvert$	$\lvert\Phi\rvert$	$\lvert\mathcal{T}\rvert$	Vulnerable?	Offline Phase Time (seconds)			
							Sym. Ex.	Noise Est.	Merging	Total
1	STAC-1(nv)	1	$2^{\{8,16,24,31\}}$	2	1	no	0.57616	22.2824	0.81452	23.67309
2	STAC-3(nv)	1	$2^{\{8,16,24,31\}}$	6	3	no	0.64201	36.1853	4.89276	41.72007
3	STAC-1(v)	1	$2^{\{8,16,24,31\}}$	2	2	yes	0.56815	31.5264	0.4860	32.5805
4	STAC-3(v)	1	$2^{\{8,16,24,31\}}$	6	4	yes	0.57977	34.0982	5.1727	39.8507
5	STAC-11A(v)	1	$2^{\{8,16,24,31\}}$	3	2	yes	0.58217	25.6543	1.3256	27.5621
6	STAC-11B(v)	1	$2^{\{8,16,24,31\}}$	3	2	yes	0.57314	26.6361	1.2993	28.5086
7	STAC-4(v)	1	26	10	2	yes	0.73373	14.7962	7.10060	22.63053
8	STAC-4(v)	2	702	27	3	yes	1.19927	44.5297	2.28398	48.01295
9	STAC-4(v)	3	18278	55	5	yes	2.67069	100.554	64.9471	168.1724
10	STAC-12(v)	1	26	17	4	yes	0.94751	26.3065	18.5799	45.83391
11	STAC-12(v)	2	702	39	5	yes	0.99083	57.4673	48.6784	107.13653
12	STAC-12(v)	3	18278	77	6	yes	1.62901	125.499	132.635	259.76301
13	STAC-12(v)	4	475254	149	7	yes	3.06844	258.488	293.578	555.13444

TABLE 2: Synthesized input strings for STAC12(v) where the secret is 'ciqa'. Partial matches indicated in **bold**.

Phase 0			Phase 1		Phase 2					Phase 3					Phase 4
prefix = ε			prefix = c		prefix = ci					prefix = ciq					prefix = ciqa
ε	fzgk	maau	**c**nte	**c**ved	**ci**ub	**ci**ij	**ci**mq	**ci**tz	**ciq**z	**ciq**i	**ciq**z	**ciq**z	**ciq**u	**ciq**z	**ciqa**
daaz	zgap	vzsc	**c**tdo	**ci**il	**ci**az	**ci**ok	**ci**da	**ci**jw	**ci**hs	**ciq**c	**ciq**z	**ciq**e	**ciq**r	**ciq**r	**ciqa**
uaak	bnza	qyas	**c**vfo	**c**eyu	**ci**gz	**ci**su	**ci**sp	**ci**ne	**ciq**k	**ciq**k	**ciq**d	**ciq**d	**ciq**r	**ciq**z	**ciq**g
ecjq	zmna	asvr	**c**sja	**ci**vf	**ci**fl	**ci**ld	**ci**cz	**ci**le	**ci**eb	**ciq**z	**ciq**q	**ciq**o	**ciq**i	**ciqa**	**ciqa**
tzar	zmna	**c**mxq	**c**wcs		**ci**kt	**ci**pa	**ci**bn	**ci**rx	**ciq**a	**ciq**s	**ciq**z	**ciq**x	**ciq**v		

STAC–3(v) is a program that contains 6 different branches and an internal parameter n which, depending on h and l, causes 2 branches to run in $O(1)$ time, 1 branch to run in $O(n)$ time, 1 branch to run in $O(n^2)$ time, and 2 branches to run in $O(n^3)$ time. The low input to the program consists of 2 values. The offline phase merged the path constraints for branches with the same time complexities into the same trace classes. The online attack phase automatically chose inputs which caused the program to always either execute the $O(n)$ or $O(n^2)$ branch in order to leverage the timing difference and fully leak the secret.

The offline phase data are shown in Table 1 and online attack phase data in Figures 10 and 11. The online attack phase was run over different secret domain sizes. For each domain size we ran until $p(H = h)$ converged with certainty to a single value, which we manually verified was the correct secret. We observe that in all cases, optimizing for trace class entropy generates an attack that takes either the same number of steps or is slightly longer. However, optimizing for trace class entropy synthesizes attack steps much more quickly. For both STAC–1(v) and STAC–3(v), we were able to synthesize attacks for domains of size 2^{31} in under 2 minutes, including offline and online phases. Because optimizing trace class entropy is significantly faster and generates strong attacks, the remaining experiments were run only with CHOOSELOWINPUT2.

Attack synthesis with programmatic non-determinism. Program STAC–11(v) has 2 versions, A and B. Both versions contain explicit recursive randomization that affects the running time of the application. In both cases, SPF is able to extract path conditions which depend only on h and l and do not depend on any randomly generated variables. Thus, as discussed in 4.7, the effect of the randomization on the running time is independent of the path conditions on h and l and can be determined using profiling during the offline phase which is presented in Table 1. In Figure 12 we see the running time and number of attack steps required to completely leak the secret for both versions of STAC–11(v).

Programs with segment oracle side channels. STAC–4(v) and STAC–12(v) are programs with segment oracle side channel [6]. Segment oracle side channels allow an attacker to use timing measurements to incrementally leak information about contiguous segments of arrays or strings. The relevant source code for STAC–12(v) is shown in Figure 14. The function under test is `verifyCredentials`, which, using the function `checkChar`, is a (somewhat obfuscated) way of comparing a candidate password to an actual secret password as part of a log-in system where valid strings consist of lowercase letters. DARPA challenge problems sometimes contain `delay` functions which mimic additional computational work. At a high level, this function compares individual characters of the secret and candidate password one at a time in a loop from left to right, and the running time of the function is proportional to the number of characters which match.

For example, if the password is 'ciqa', the running time will be slightly longer if an attacker inputs 'ciqg' (first 3 characters match) vs. 'ciao' (first 2 characters match). Thus, an attacker can use a timing side channel to reveal prefixes of the secret, and reduce the $O(k^n)$ brute-force search space to $O(k \cdot n)$. Segment oracle attacks were responsible for several real-world vulnerabilities [23], [1], [2].

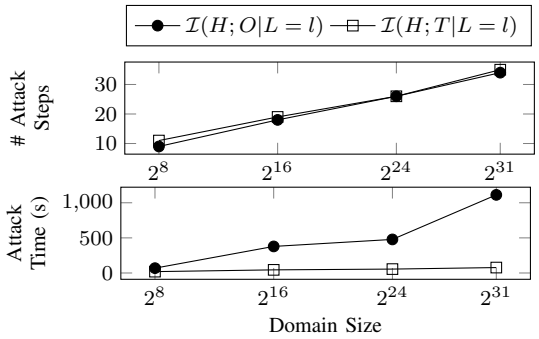


Figure 10: STAC–1(v): Comparing attacks generated by optimization based on observation vs. trace class entropy.

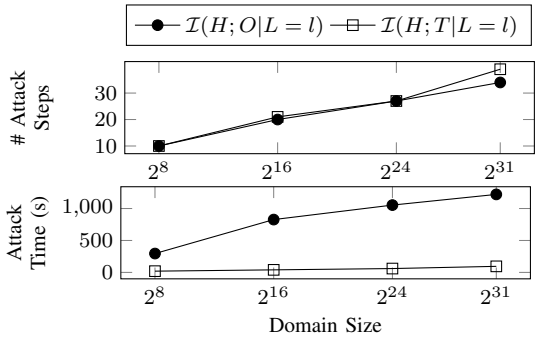


Figure 11: STAC–3(v): Comparing attacks generated by optimization based on observation vs. trace class entropy.

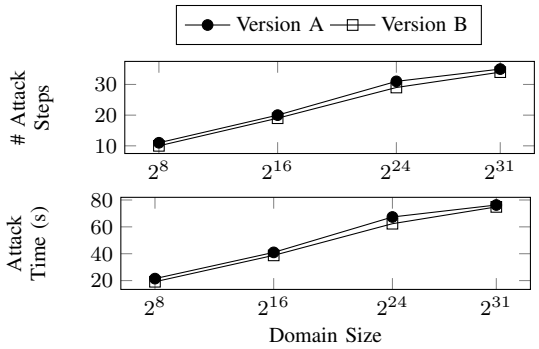


Figure 12: STAC–11(v). Two versions, A and B.

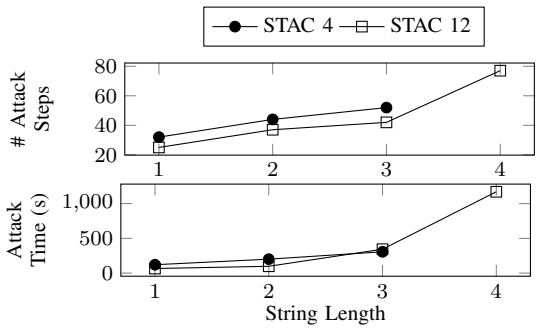


Figure 13: STAC–4(v) and STAC–12(v)

```
 1 private static String password;
 2 private static int subsequentCorrect;
 3 private static int exceedPasswordLen;
 4 private static void delay() {
 5     for (int x=0 ; x < 75000 ; x++) {}
 6 }
 7 private static void
     checkChar(String candidate, int charNumber){
 8    if(charNumber > password.length())
 9        exceedPasswordLen++;
10    else if(password.charAt(charNumber - 1) ==
              candidate.charAt(charNumber - 1)){
11         if(subsequentCorrect+1 == charNumber){
12             subsequentCorrect++;
13             delay();
14         }
15     }
16 }
17 private static boolean
     verifyCredentials(String candidate){
18    subsequentCorrect = exceedPasswordLen = 0;
19    for (int x=0; x < candidate.length(); x++){
20        checkChar(candidate,x+1);
21    }
22    return subsequentCorrect == password.length()
            && exceedPasswordLen == 0;
23 }
```

Figure 14: Relevant source code of STAC-12(v).

Our method automatically synthesizes segment oracle attacks. We ran our attack synthesis method on STAC–4(v) and STAC–12(v) where the secret domain is strings of lowercase letters. We set an online phase timeout of 20 minutes and ran attack synthesis for increasing string lengths. We generated attack steps for STAC–4(v) up to string length 3 before timing out. For STAC–12(v) we generated attack steps up to string length 4. The offline phase data for STAC–4(v) and STAC–12(v) are shown in Table 1 and online phase data is shown in Figure 13.

We give some details of one synthesized attack. For STAC12(v), length 4 (Table 1, benchmark 13) the secret was a randomly generated string, 'ciqa'. In Table 2 we manually separated the synthesized input strings into phases where the i^{th} phase roughly corresponds to inputs that match a secret prefix of length i. For instance, in phase 0, the attacker does not know any prefix of the secret password, first tests the system with ε, then begins testing inputs with different characters. At the end of phase 0, the attacker has discovered using side-channel measurements that the first character is 'c'. Thus, in phase 2, the attacker keeps the first character 'c' constant and tests other inputs until the side-channel observation indicates that the first 2 characters, 'ci', match. This continues similarly for phases 3 and 4, until all characters are discovered. This required 77 steps overall to discover a secret from a search space of size 475254.

Observe that some strings were tested multiple times. For instance, the string 'ciqz' was synthesized several times over Phases 2 and 3, and the final secret 'ciqa' was tested several times during the last phase. Thus, we observe that our online attack synthesis technique appears to automatically discover the fact that repeated sampling can be used to eliminate spurious observations due to noise and reduce uncertainty. Indeed, real-world manually written attacks often

employ repeated sampling in an attempt to eliminate noise.

Finally, note that, once our automated attack synthesis approach generates a segment oracle attack for a program for a small secret length, it is easy to generalize such an attack to larger secrets by inspecting the generated attack.

5.3. Case Study: Law Enforcement Database

We applied our method to a larger application provided by DARPA. LawDB (Law Enforcement Database) is a network service that stores and manipulates a database used to store law enforcement personnel data associated with user IDs. Users can issue the command `SEARCH minID maxID` to query the database for IDs within a range. IDs are internally stored as public or restricted. Only public IDs within the search range will be shown to the user; restricted IDs are secret. Using our approach, we were able to synthesize a timing side-channel attack for this application that enables a public user to determine a restricted ID.

Symbolically executing the entire LawDB application (51 classes, 202 methods) was not feasible. By examining the source code, it was straightforward to locate the method for the `SEARCH` operation, which corresponds to case 8 of the method `channelRead0` of class `UDPServerHandler` (Figure 15). We noticed a possible side channel because the UDP request handler writes the message "`SEARCH ON RESTRICTED KEY OCCURRED`" to a log file and throws a `RestrictedAccessException` depending on whether a user has entered a search query range which encompasses a restricted secret ID.

We extracted the `UDPServerHandler` class (Figure 15) and its closure of dependencies from the source code. Symbolic execution was not able to handle the provided log writing, so we supplied our own simplified code which writes the same message to a log file in order to mimic the original behavior of the function. We wrote a small driver (20 LOC) to interface our symbolic execution system and profiling server with the `UDPServerHandler` class. The driver initializes the database by inserting n unrestricted IDs and inserts one restricted ID, h. The driver then executes queries with range: `SEARCH` l_{minID} l_{maxID}. We symbolically execute the driver with h, l_{minID}, and l_{maxID} all symbolic and then profile the driver with the generated witnesses to conduct the offline phase. Then our online attack phase automatically synthesizes adaptive `SEARCH` range queries that eventually reveal the restricted ID.

Initial Exploratory Experiment. We initialized the database with two concrete unrestricted IDs (with values 64 and 85) and constrained allowed IDs to the range $[1, 100]$. The result of running attack synthesis on this configuration can be seen in Table 3 and intermediate snapshots of online attack inputs and belief updates are shown in Figure 16. We see that there are 42 path constraints generated by symbolic execution which reduce to 3 trace classes after profiling and merging. The offline phase takes less than 1 minute and the online attack phase of 25 steps takes less than 3 minutes.The secret ID was set to be a randomly generated number, 92.

Figure 16 illustrates interesting self-correcting behavior of the automated search. Due to observation noise, in step 5 a timing observation was made that caused the belief distribution to become highly concentrated in the range $[64, 76]$. The subsequently synthesized inputs are queries which search this interval and eventually eliminate that concentration of probability mass. Steps 12 through 25 generate queries that concentrate the belief on the secret ID, 92.

Larger Experiments. After seeing that we can automatically synthesize an adaptive range-query attack against LawDB for small domains, we increased the valid range of IDs that are allowed in the database to $[1, 10000]$ and inserted 3, 4, and 8 randomly generated public IDs. In Table 3 we see that we are able to synthesize attacks in a reasonable amount of time. We observe that as the number of path constraints increases, the cost of the offline phase grows, with the majority of offline time spent in trace class merging. However, trace class merging is crucial since it reduces the difficulty and the cost of model counting and entropy computations. For instance, for LawDB-4 model counting and entropy computation may be performed over 9 trace classes rather than 855 path constraints, resulting in efficient online attack synthesis.

6. Related Work

There is a considerable amount of fairly recent work related to quantifying information leakage for a single run of a program [5], [18], [31], [21], [28], [30], [29], but none of these considers multiple runs of the program. Work in [32] addresses computing side-channel leakage for multiple runs of a program using symbolic execution but does not address adaptively chosen input sequences.

The most closely related work in this area uses a fully static approach for synthesizing adaptive attack strategy trees over all possible secrets [27]. This earlier approach makes use of MaxSMT and Barvinok for choosing inputs, but (i) since it tries to generate the complete attack tree statically for all possible secrets, it is not scalable to generation of attacks on realistic systems, and (ii) it does not take system noise into account. Note that precomputing the attack tree for all possible secrets is always exponential in the length of the attack. Additionally, in this earlier approach each Java bytecode is modeled to have the same contribution to side-channel observations, which is not a realistic assumption. Moreover, an observation abstraction is used to merge path constraints with similar observations, which uses a naive manual discretization. Other closely related work also assumes noiselessness, does not handle weighted belief updates, precomputes a full attack tree, and uses explicit (non-symbolic) domain representation [22].

Hence, our key contributions with respect to the most closely related works [22], [27] are: 1) handling noise in observations, 2) weighted belief update via weighted model counting, accounting for the probabilistic nature of observations, 3) online attack generation without generating a complete attack tree, 4) automated identification of trace classes

```
 1 public class UDPServerHandler {
 2 // Constructors and local variables ...
 3
 4 public void channelRead0(int t, int key,
 5     int min, int max) {
 6   switch (t) {
 7   ...
 8   case 8: {
 9    DSystemHandle sys = new
10    DSystemHandle("127.0.0.1", 6666);
11    List<String> filestoCheck=new ArrayList<String>();
12    final DFileHandle fh1 =
13      new DFileHandle("config.security", sys);
14    filestoCheck.add("config.security");
15    final List<Integer> range =
16      this.btree.toList(min, max);
17    if (range.size() <= 0 || !this.restricted.
18        isRestricted((int) range.get(0))) {
19      filestoCheck = new ArrayList<String>();
20    }
21    int ind = 0;
22    while (ind < range.size()) {
23      try {
24        final Integer nextkey=(Integer)range.get(ind);
25        if (this.restricted.isRestricted(nextkey)) {
26          BufferedWriter bw = null;
27          FileWriter fw = null;
28          try {
29            String data =
30              "SEARCH ON RESTRICTED KEY OCCURRED:" +
31              (Integer.toString((int) nextkey) + "\n");
32            File file = new File(LOGFILE);
33            if (!file.exists()) {
34              file.createNewFile();
35            }
36            fw = new FileWriter(file.getAbsoluteFile(),
37               true);
38            bw = new BufferedWriter(fw);
39            bw.write(data);
40          }
41          catch (IOException e) {
42            e.printStackTrace();
43          }
44          finally {
45            try {
46              if (bw != null)
47                bw.close();
48              if (fw != null)
49                fw.close();
50            } catch (IOException ex) {
51              ex.printStackTrace();
52            }
53          }
54          throw new RestrictedAccessException();
55        }
56        if (sys == null) {
57          sys = new DSystemHandle("127.0.0.1", 6666);
58        }
59        at.add("lastaccessinfo.log",
60          Integer.toString(nextkey), nextkey);
61        ++ind;
62      } catch (RestrictedAccessException rae) {
63        for (Integer getkey = (Integer) range.get(ind);
64          this.restricted.isRestricted(getkey)
65            && ind < range.size();
66            getkey = (Integer) range.get(ind)) {
67          if (sys == null) {
68            sys = new DSystemHandle("127.0.0.1", 6666);
69          }
70          if (++ind < range.size()) {
71          }
72        }
73      } finally {
74        if (atx != null) {
75          System.out.println("Cleaning resources");
76          atx.clean();
77          atx = null;
78        }
79      }
80    }
81    at.clean();
82    sys = new DSystemHandle("127.0.0.1", 6666);
83    break;
84  }
85
86  // Remaining switch cases...
```

Figure 15: Extracted search function for LawDB.

TABLE 3: Experimental data for 4 different instantiations of the LawDB case study.

						Offline Phase Time(s)				Online Phase	
Benchmark		# IDs in DB	$\lvert\mathbb{H}\rvert$	$\lvert\Phi\rvert$	$\lvert\mathcal{T}\rvert$	Sym. Ex.	Noise Est.	Merging	Total	Time(s)	# Steps
14	LawDB-1	3	100	42	3	1.17	5.45	51.10	57.736	158.78	25
15	LawDB-2	4	10000	90	4	1.81	11.83	127.59	141.24	163.28	45
16	LawDB-3	5	10000	165	5	2.91	23.39	365.13	391.45	188.85	48
17	LawDB-4	9	10000	855	9	20.57	152.09	2436.84	2609.5	271.16	77

from noisy observations using profiling and automatic path constraint merging.

A model of belief update for information flow to an adversary who makes observations of a probabilistic program was presented in [12]. Their work specifically addresses single runs of a program, but the authors indicate that Bayesian belief updates may be used over multiple runs of the program. In [9], the authors illustrate methods for detecting the possibility of side channels in client-server application traffic. Work in [4] also addresses DARPA STAC programs. Their effort is concentrated on showing safety properties of non-vulnerable programs and is able to indicate possible side-channel vulnerabilities by detecting observationally imbalanced program branches, but does not generate attacks. Two works [24], [3] describe how to quantify information leakage in interactive systems in which the secret changes over repeated runs of the application. In [25] a method is given to quantify the trade-off between program reliability and security when adding noise to a program in order to reduce side-channel vulnerability. In [6] a method is given to quantify the vulnerability of a program to a given segment attack, similar to that of STAC–4(v) and STAC–12(v) for which our work synthesizes an attack. In [11] the problem of performing Bayesian inference via weighted model counting is described with applications to statistical prediction. In [20] a method is given for performing precise quantitative information flow analysis using symbolic weighted model counting. Improving scalability of our approach is important, although we note that the state-of-the-art techniques for synthesizing or analyzing adaptive attacks [6,21,29,31,32]

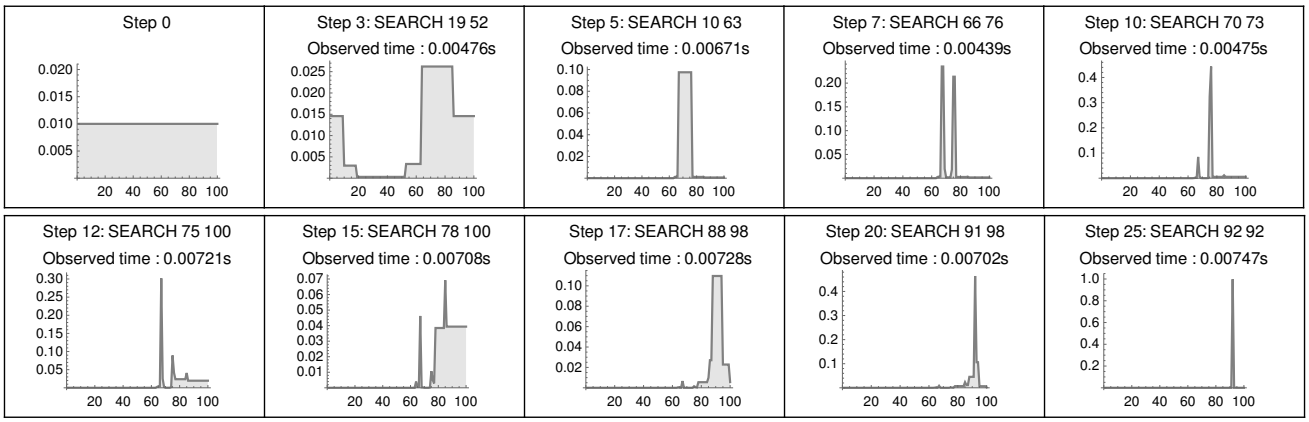


Figure 16: Example snapshots of $\mathcal{A}$'s belief about the restricted ID for the LawDB-1 case study. In step 5, a noisy observation causes an erroneous belief update, but the synthesized queries eliminate the incorrect belief, to eventually converge to the true value of the restricted ID, 92.

do not handle noise, do not synthesize attacks against a live server, and yet, still, are not more scalable than our approach.

7. Conclusion

We presented a technique to automatically generate online side-channel attacks and we experimentally showed that we are able to adaptively synthesize a sequence of inputs to discover the secret via observations in a noisy environment. We leverage static code analysis using symbolic execution and offline dynamic profiling. We use symbolic weighted model counting for performing Bayesian inferences about the secret value stored in the program and we cast the problem as an entropy objective function maximization problem which we solve using numeric optimization.

There are several avenues for future work in adaptive attack synthesis. Our method uses constraints to characterize trace classes. We employ symbolic execution to generate such constraints, but other methods can also be used (abstract interpretation, invariant generation, software model checking) and we intend to investigate and compare them. We observed that programs may have many path constraints, but few trace classes, and so any technique which can more directly produce trace classes without merging finer partitions (like path constraints) will yield improvement. Thus, our overall approach is not constrained by the limitations of symbolic execution, only by the ability to discover trace classes. Additionally, incorporating dynamic techniques like fuzzing [39] for trace class discovery, in order to move toward a more black-box or gray-box approach, can supplant the need to use symbolic execution and constraint solving, thereby reducing the overhead of the static offline phase.