Data-Driven Debugging for Functional Side Channels

Saeid Tizpaz-Niari
University of Colorado Boulder
saeid.tizpazniari@colorado.edu

Pavol Černý
TU Wien
pavol.cerny@tuwien.ac.at

Ashutosh Trivedi
University of Colorado Boulder
ashutosh.trivedi@colorado.edu

***Abstract*—Information leaks through side channels are a pervasive problem, even in security-critical applications. *Functional side channels* arise when an attacker knows that a secret value of a server stays fixed for a certain time. Then, the attacker can observe the server executions on a sequence of different public inputs, each paired with the same secret input. Thus for each secret, the attacker observes a (partial) function from public inputs to execution time, for instance, and she can compare these functions for different secrets.**

First, we introduce a notion of noninterference for functional side channels. We focus on the case of noisy observations, where we demonstrate with examples that there is a practical functional side channel in programs that would be deemed information-leak-free or be underestimated using the standard definition. Second, we develop a framework and techniques for debugging programs for functional side channels. We extend evolutionary fuzzing techniques to generate inputs that exploit functional dependencies of response times on public inputs. We adapt existing results and algorithms in functional data analysis (such as functional clustering) to model the functions and discover the existence of side channels. We use a functional extension of standard decision tree learning to pinpoint the code fragments causing a side channel if there is one.

We empirically evaluate the performance of our tool FUCHSIA on a series of micro-benchmarks, as well as on realistic Java programs. On the set of micro-benchmark, we show that FUCHSIA outperforms the state-of-the-art techniques in detecting side-channel classes. On the realistic programs, we show the scalability of FUCHSIA in analyzing functional side channels in Java programs with thousands of methods. In addition, we show the usefulness of FUCHSIA in finding (and locating in code) side channels including a zero-day vulnerability in Open Java Development Kit and another Java web server vulnerability that was since fixed by the original developers.

I. Introduction

Developers are careful to assure that eavesdroppers cannot easily access the secrets by employing security practices such as encryption. However, a side channel might arise even if the transferred data is encrypted. The side-channel eavesdroppers can infer the value of secret inputs (or some of their properties) based on public inputs, runtime observations, and the source code of the program. An example is OnlineHealth service [1], where the service leaks the conditions of patients through side channels observable in the characteristics of network packets.

Network and Distributed Systems Security (NDSS) Symposium 2020
23-26 February 2020, San Diego, CA, USA
ISBN 1-891562-61-4
https://dx.doi.org/10.14722/ndss.2020.24269
www.ndss-symposium.org

We consider *known-message* threats [2] where the attacker knows the value of public inputs as well as execution times when trying to find out the secret. In this threat model, we consider the setting where the secret input stays fixed across a number of interactions. This gives rise to *functional observations*: for a secret input, we observe the program executions on a number of public inputs. For a secret input s, we obtain a partial function f_s from public inputs to runtime observations. We focus on timing side channels, where the attacker's observations are the execution time.

Functional side channels. We adapt the classical definition of noninterference to *functional side channels*, where two secret inputs s and t are indistinguishable for the attacker if the functions f_s and f_t are equal. However, in the presence of noise (a common situation for timing measurements), we cannot require exact equality of functions. Instead, we define two functional observations to be indistinguishable when they are similar according to a notion of distance. We demonstrate on a set of examples that it is critical to choose the distance that represents functional observations, otherwise, side channels might remain undetected or be underestimated.

Problem. *Data-driven debugging focuses on automatically discovering functional timing side channels, and on pinpointing code regions that contribute to creating the side channels.*

Algorithms. As functional timing side channels are hard to detect statically with the current program analysis tools, we turn to dynamic analysis methods. We propose to use gray-box evolutionary search algorithms [3], [4] to generate interesting secret and public inputs. We use *functional data clustering* [5], [6] to model functional observations, discover timing side channels, and estimate their strengths. It allows us to compute an equivalence relation on secret inputs that model the distinguishing power of the attacker. If this relation has multiple equivalence classes, there is an information leak. In order to find what parts of the code caused the leak, we identify features that are common for secrets in the same cluster (equivalence class), and features that separate the clusters. Typical features in the debugging context are program internals such as methods called or basic blocks executed for a given secret value. We present functional extensions to *decision tree inference techniques* to locate code regions that explain differences among clusters. These code regions are thus suspect of being a root cause of the side channels.

Experiments. We evaluate our tool FUCHSIA on micro-benchmarks and 10 larger case studies. We use micro-benchmarks to evaluate the scalability of components in our tool and compare FUCHSIA to the state-of-the-art. The case studies serve to evaluate scalability and usefulness on real-world Java applications.

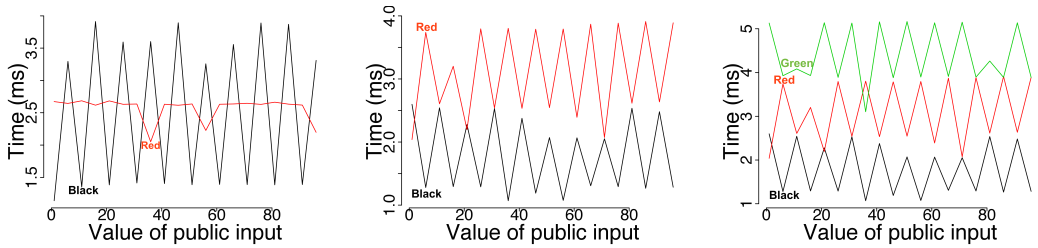


Fig. 1. (a) Functional observations. (b) Attacker's local observations. (c) Attacker's remote observations.

Contributions. Our main contributions are:

- Defining *functional noninterference* in the presence of noisy observation: We demonstrate functional side channels caught by our definitions in programs that would be deemed information-leak-free or underestimated using the standard (non-functional) definition.
- Algorithms: We adapt existing theory and algorithms for *functional data clustering* to discover the existence of side channels. We develop a *functional extension of decision tree learning* to locate the code regions causing a side channel if there is one.
- Experiments: we show on micro-benchmarks that FUCHSIA outperforms DifFuzz [7], a state-of-the-art technique, in quantifying the strength of leaks using the number of classes in timing observations.
- Case Studies: We show the scalability of FUCHSIA in analyzing Java programs with thousands of methods. FUCHSIA finds a zero-day vulnerability in OpenJDK and vulnerability in Java web-server that was since fixed by the original developers.

II. FUNCTIONAL SIDE CHANNELS

We first illustrate what an attacker can infer based on functional observations, even in the presence of noise. Second, we show that it is critical to use functional observations to evaluate the resulting threats of side channels.

A. Functional observations and timing side channels

We consider the *known-message* threats [2] where the attacker knows public values, but she cannot control them. We focus on the situation where secret values remained unchanged for some amount of time (e.g., passwords, social security number, and friends of a user in social media). The attacker who has access to the source code tries to infer the secret by observing the execution time and knowing the public values.

Let us consider the classical definition of confidentiality:– noninterference. A program is *unsafe* iff for a pair of secret values s_1 and s_2, there exists a public value p such that the behavior of the program on (s_1, p) is observably different than on (s_2, p). If our observable is the execution time T, then the program is unsafe iff: $\exists s_1, s_2, p : T(s_1, p) \neq T(s_2, p)$. In our setting, for each secret value, we observe the execution time of a program on a number of public values. Thus, the program is unsafe iff: $\exists s_1, s_2 : (\lambda p.T(s_1, p)) \neq (\lambda p.T(s_2, p))$. In other words, the program is unsafe if the two secret values do not correspond to the same function of public inputs.

Side channels in the presence of noise. Quantitative observations of a program's runtime behavior are often noisy. For instance, running a program with the same inputs twice on the same machine result in different measurements of execution time. Observing the program remotely adds a further level of noise. Classical definitions of confidentiality properties, therefore, need to be adapted to noisy environments. In the noisy environment, no two observations are equal and our definition needs to include ε tolerance:

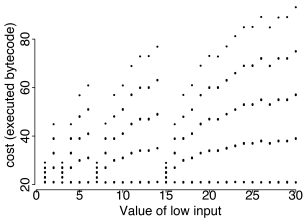
$$\exists s_1, s_2 : d(\lambda p.T(s_1, p), \lambda p.T(s_2, p)) > \epsilon. \quad (1)$$

In this definition, d is a distance between two functions. The distance is suitably chosen, typically based on the noise expected for a particular use case.

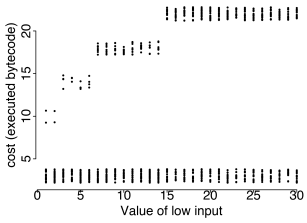
A straightforward extension of classical noninterference with ϵ tolerance to our functional setting is to use the ∞-norm for the distance function: $d_\infty(f_{s_1}, f_{s_2}) = \sup_p |f_{s_1}(p) - f_{s_2}(p)|$, where $f_s(p) = \lambda p.T(s, p)$. However, we now demonstrate that the point-wise distance d_∞ is not the only option, and that depending on the type of noise, different distances are needed. In particular, we show that if we use the point-wise distance, we could certify a side-channel vulnerable program.

Gaussian noise (pointwise independent, mean 0). Consider the two functional observations (red and black) of a program in Figure 1a. The red function corresponds to the secret value s_1 and the black function corresponds to the secret value s_2. The eavesdropper can produce this graph easily by trying possible inputs on their machine beforehand. At runtime, the eavesdropper collects the public inputs and the execution time, and tries to learn the secret by matching the observed data to the red or black functions. In this example, we assume that the noise for each pair of public-secret inputs is independent and identically distributed. Furthermore, we assume that it is distributed according to a Gaussian distribution with mean 0. Let us consider ε of 3ms, and then apply our definition with distance d_∞. We see that the two functional observations are ε-close for this distance, so the attacker cannot infer the secret value (s_1 or s_2). However, the functional observations are clearly very different, and an eavesdropper can reliably learn the secret. This can be captured using the L_1-norm as the distance function. This example shows the point-wise distance d_∞ may not be appropriate to detect certain side channels.

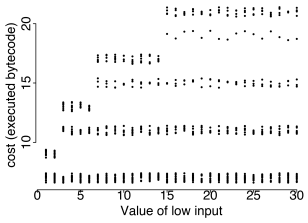
Gaussian noise (pointwise independent, mean C). Let us consider the case where the noise is Gaussian, but with a non-zero mean. The non-zero mean is fixed, but is unknown to the attacker. This case arises if, for instance, the eavesdropper is remote and cannot determine the delay introduced by the network and separate it from the noise of the remote machine.



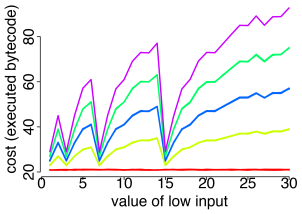
(a) Point-wise model $\mathcal{P}_1$.



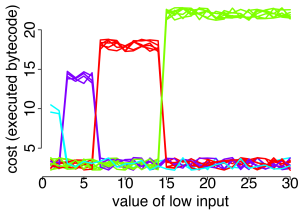
(b) Point-wise model $\mathcal{P}_2$.



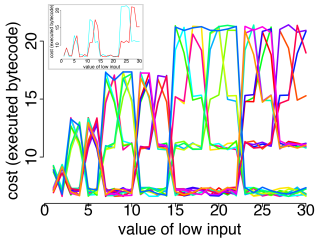
(c) Point-wise model $\mathcal{P}_3$.



(d) Functional model $\mathcal{P}_1$.



(e) Functional model $\mathcal{P}_2$.



(f) Functional model $\mathcal{P}_3$.

x-axis	low input value
1	"0"
2	"1"
3	"00"
4	"01"
5	"10"
6	"11"
7	"000"
8	"001"
...	...
14	"111"
15	"0000"
16	"0001"
...	...
30	"1111"

Fig. 2: Programs $\mathcal{P}_1$, $\mathcal{P}_2$, $\mathcal{P}_3$ are leaking the number of set bits, the length of secret, and the secret values, respectively. In programs $\mathcal{P}_2$ and $\mathcal{P}_3$, the clustering over functions are required to find the correct number of classes of observations.

Consider a program with two functional behaviors (red and black) pictured in Figure 1b, where the red and black behaviors correspond to secret s_1 and secret s_2, respectively.

At runtime, the attacker interacts with the remote server running the same instance of the application with a fixed secret value. The green timing function in Figure 1c shows the execution-time function of public inputs obtained from observing the remote server. The green function looks far apart from both local observations (black and red functions). However, due to the effects of remote observations, the attacker knows that the execution time is off by an unknown constant. The attacker is in effect observing only the shapes of functions, i.e., their first derivatives. So, the distance is over the derivatives of functions [6], [8]. The attacker can use this distance and calculate that the green function is closer to the black function than the red function. Note that if the L_1 distance between the first derivative of two timing functions is greater than ε, the corresponding secret values can leak to a remote attacker.

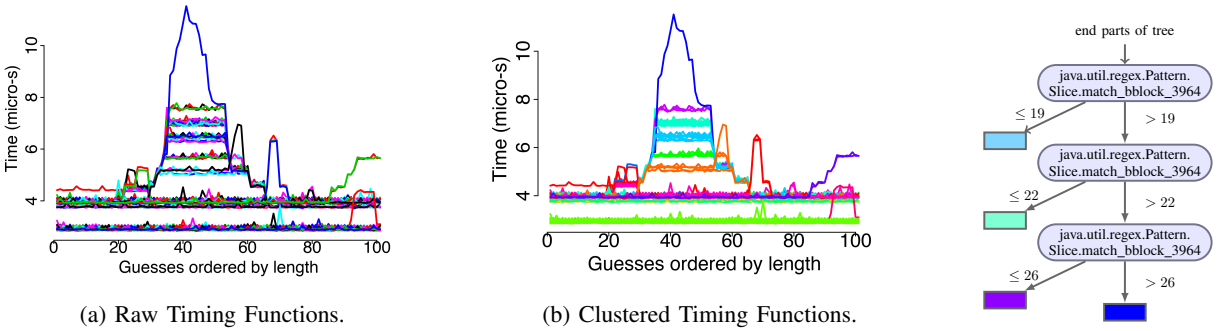
B. Classes of observations in side channels

The number of distinct timing observations (or clusters) over secret inputs is an important measure to evaluate the resulting threats of side channels [9], [2]. Here, we illustrate that it is critical to analyze functions to obtain clusters, especially in dynamic analysis. We consider three side-channel vulnerable programs. Program $\mathcal{P}_1$, a variant of square and multiply algorithm [10], leaks the number of set bits in the secret. Program $\mathcal{P}_2$, a vulnerable Jetty password matcher [11], leaks the length of secret passwords. Program $\mathcal{P}_3$, a vulnerable google Keyczar password matcher [12], leaks the value of secret passwords. Let public values be the sequence $\langle$"0", "1", "00", "01", ..., "1111"$\rangle$, and secret values be the set $\{$"0", "1", "00", "01", ..., "1111"$\}$.

Figure 2 shows six plots about the execution times of three programs: Figure 2 (a-c) are point-wise depictions and Figure 2 (d-f) are functional presentations. The x-axis is the index of 30 public values (see corresponding values in the table in Figure 2). The y-axis is the cost of executing a pair of secret and public values in the number of bytecode executed.

According to the point-wise definition, the number of clusters can be obtained by fixing the public input and finding the number of distinguishable classes of observations (different cost values) over all possible secrets. Let's consider a public input value that gives the largest number of clusters for each example. Let's pick the index 30 on the x-axis for all examples in Figure 2 (a-c). This choice results in 5, 2, and 5 clusters for programs $\mathcal{P}_1$, $\mathcal{P}_2$, and $\mathcal{P}_3$, respectively. According to the functional definition, we model the execution times of each secret value as a function from the public input to the cost of execution. Figure 2 (d-f) show 30 functions in each plot, colored based on their cluster labels. Any two functions that are $\epsilon = 0.1$ close to each other belong to the same cluster.

For $\mathcal{P}_1$, there are 5 clusters in Figure 2 (a), and also 5 functional clusters in Figure 2 (d). This means that both the point-wise and functional definitions agree that there are five clusters (0 to 4 possible set bits). However, the results are different for programs $\mathcal{P}_2$ and $\mathcal{P}_3$. Figure 2 (b) shows that in each point on the public value, there are two classes of observations (either the lengths of secret and public inputs matched or not). On the other hand, the functional model in Figure 2 (e) shows that there are 4 different functional clusters that correspond to four possible lengths in the secret values. Specifically, let's look at Figure 2 (e) from the indices 7 to 14. These are the low (public) values with the length of three (from "000" to "111"). Secret values with length three are checked against these public values, while all other secret values are rejected immediately. As a result, the timing functions of secret values with the length three goes up from 2 to 17 at index 7 and goes down from 17 to 2 at index of 15. Observe that the clustering helps to group timing functions of secret values with length three in the same class. Similarly, there are unique indices for secrets with the lengths 1, 2, and 4 where the timing functions jump up/down and reveal the length of secrets.



(a) Raw Timing Functions.

(b) Clustered Timing Functions.

Fig. 3: (a) Regex timing functions. Execution times of 435 secret values are modeled as 435 timing functions. How are these functions related? (b) 435 timing functions are clustered into 12 distinguishable classes of observations (clusters) using L_1-norm distances. The presence of different clusters indicate some properties of secret patterns are leaking. What properties of secret patterns are leaking? (c) Regex decision trees. The number of calls to the basic block at line 3964 of `Pattern.match()` method (shown in Figure 4) discriminates different clusters. The code region shows the value of secret patterns is leaking.

Similarly, the point-wise model for the program $\mathcal{P}_3$ in Figure 2 (c) underestimates that the (whole) secret values are leaking (found 5 clusters). The functional model in Figure 2 (f) finds 30 clusters: there is a unique function for each of the 30 secret values (two samples are shown on top of Figure 2(f)).

III. Overview

We illustrate how our tool can be used for discovering and explaining a zero-day timing side channel. We analyze the `java.util.regex` [1] package of Open JDK 8. The package includes 620 methods and over 8,000 lines of code.

Problem. The secret input is the regular expression compiled as a pattern, and the public input is the guess that matches against the pattern. The attacker's goal is to infer a (fixed) secret (or its properties) by observing the response time for multiple public inputs. The key problem is to help the *defender* discover the existence of such side channels.

Side channel discovery. The defender starts choosing a finite set of secret and public values. In this example, the defender uses FUCHSIA to generate the set of secret and public input values. The fuzzing of FUCHSIA is an extension of AFL [3] and Kelinci [4] where it generates multiple public input values for each secret value to discover the functional dependencies of response time over public inputs. The defender obtains 1,154 different secret patterns and 6,365 different matching guesses during 4 hours of fuzzing. The lengths of guesses are at most 52 characters. For each secret value, FUCHSIA varies 6,365 different guesses and measures the execution time of regex. Then, it fits B-spline [13] to model the timing functions for each secret. Figure 3 (a) shows 435 different timing functions over the public inputs from the guesses with the prefix "bb..." to "gg...". We choose the subsets of secret and public values for simpler presentations. Next, the defender wants to know how these timing functions are related and if there are timing side channels.

The defender provides the notion of a distance and the tolerance ε. In this case, he considers L_1-norm distance between functions and the tolerance $\varepsilon = 0.2$. Given the distance norm and the tolerance, FUCHSIA uses a non-parametric functional clustering algorithm [5] to discover classes of observations. The clustering algorithm finds 162 clusters over the $1,154$ secret values. The existence of 162 distinct classes of observations indicates the presence of a functional side channel in the regex package. Figure 3 (b) shows 12 clusters for the subset of 435 functions presented in Figure 3 (a).

Side channel explanation. Now, the defender wants to know what properties of secret are leaking through the timing side channels. The task is to learn the discriminant [14]. It helps the defender localize the code regions causing to observe different clusters and use the information to establish facts about the leaks. Specifically, it shows which features are common for secrets in the same cluster and which features separate different clusters. FUCHSIA uses program internal features such as methods called or basic blocks invoked. These are obtained by running the same set of secret and public inputs on the instrumented regex (using Javassist [15]). The instrumentation provides 203 features about the internals of regex. FUCHSIA applies an extension of the decision tree learning algorithm from [16]. It produces a decision tree whose nodes are labeled by program internal features and whose leaves represent sets of secret values inside a cluster.

Figure 3 (c) shows (parts of) the decision tree model learned for regex. Using this model, the defender realizes that the executions of the basic block at line 3964 of `Pattern`

Regex.Pattern.Slice

```
boolean match(Matcher matcher, int i, ...
  CharSequence seq){
    int[] buf = buffer;
    int len = buf.length;
    for (int j=0; j<len; j++) {    (line.3964)
      if ((i+j) >= matcher.to){
        matcher.hitEnd = true;
        return false;
      }
      if (buf[j] != seq.charAt(i+j))
        return false;
    }
    return next.match(matcher, i+len, seq);
  }
```

Fig. 4: Pattern matching using regex (`buf` secret, `seq` public).

[1] https://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html

class in `Slice.match()` method is what distinguishes the clusters. This basic block represents the loop body of the `for` statement in the method shown in Figure 4. For instance, the purple cluster, the functions with the second highest execution times around the index 40 of Figure 3 (b), corresponds to the case where `match_bblock_3964` calls more than 22 times but less than 26 times. Note that the edge values in the decision tree are also B-spline functions, but we map them to their max values for the illustration. Inspecting the relevant code, the defender realizes that the matching prefix of secret patterns is leaking through the side channels. Hence, an attacker can obtain the (whole) secret patterns, one part in each step of observations. We reported this vulnerability, and the OpenJDK security team has confirmed it. Since fixing this vulnerability requires substantial modifications of library, the developers suggested to add artificial extra delays when the stored pattern is secret in order to mitigate it.

IV. Definitions

We develop a framework for detecting and explaining information leaks due to *functional timing observations*.

A. Threat Model

We consider the known-message threat [2] and assume that the secret inputs are less volatile than public inputs. Thus, the attacker's observations are functional where for each secret value, she learns a function from the public inputs to the execution times. In our threat model, the attacker, who has access to the source code, can sample execution times arbitrarily many times on her local machine with different combinations of secret and public values. She can thus infer an arbitrarily accurate model of the application's execution times. During the observations on the target machine, the attacker intends to guess a fixed secret by observing the application on multiple public inputs. These observations may not correspond to her local observations due to several factors, such as, i) target's machine noises, ii) network delays, and iii) masking delays added to every response time to mitigate side channels.

B. Timing Model and Functional Observations

Let $\mathbb{R}$ and $\mathbb{R}_{\geq 0}$ be the set of reals and positive reals. Variables with unspecified types are assumed to be real-valued.

Definition IV.1. *The timing model $[\![\mathcal{P}]\!]$ of a program $\mathcal{P}$ is a tuple (X, Y, Σ, δ) where:*

- *$X = \{x_1, \ldots, x_n\}$ is the set of secret-input variables,*
- *$Y = \{y_1, \ldots, y_m\}$ is the set of public-input variables,*
- *$\Sigma \subseteq \mathbb{R}^n$ is a finite set of secret-inputs, and*
- *$\delta : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}_{\geq 0}$ is the execution-time function of the program as a function of secret and public inputs.*

A *functional observation* of the program $\mathcal{P}$ for a secret input $s \in \Sigma$ is the function $\delta(s)$ defined as $\mathbf{y} \in \mathbb{R}^m \mapsto \delta(s, \mathbf{y})$. Let $\mathcal{F}$ be the set of all functional observations. To characterize indistinguishability between two functional observations, we introduce a (normalized) distance function $d_{i,p} : \mathcal{F} \times \mathcal{F} \to \mathbb{R}_{\geq 0}$ on functional observations, for $i, p \in \mathbb{N}$, defined as:

$$d_{i,p} \stackrel{\text{def}}{=} (f, g) \mapsto \left(\frac{1}{|Y|} \int_{y \in Y} \left(f^{(i)}(y) - g^{(i)}(y) \right)^p dy \right)^{\frac{1}{p}},$$

where $f^{(i)}$ represents i-th derivative (wrt y) of the function f (0-th derivative is the function itself) and $|Y|$ is a measure for the domain of public inputs. The distance function $d_{i,p}$ corresponds to the p-norm distance between i-th derivatives of the functional observations. Given the tolerance $\varepsilon > 0$ and the distance function parameterized with $i, p \in \mathbb{N}$, we say that secrets s and s' are indistinguishable if $d_{i,p}(\delta(s), \delta(s')) \leq \varepsilon$.

Depending upon the context, as we argued in Section II-A, different distance functions may be of interest. For instance, the distance between first derivatives may be applicable when the shape of the functional observation is leaking information about the secret and the second derivatives may be applicable when the number of growth spurts in the observations is leaking information. Similarly, in situations where the attacker knows the mitigation model—say, temporal noises added to the signal are n-th order polynomials of the public inputs—two functional observations whose n-th derivatives are close in the p-norm sense may be indistinguishable to the attacker. Finally, depending upon the specific situation, an analyst may wish to use a more nuanced notion of distance by taking a weighted combination [17] of various distance functions characterized by $d_{i,p}$. To keep the technical discourse simple, we will not introduce such weighted combinations.

C. Noninterference and Functional Observations

Noninterference is a well-established [18], [19], [20] criterion to guarantee the absence of side channels. A program $\mathcal{P}$ is said to satisfy the *noninterference property* if: $\forall \mathbf{y} \in \mathbb{R}^m \forall s, s' \in \Sigma$ we have $\delta(s, \mathbf{y}) = \delta(s', \mathbf{y})$. To account for the measurement noises in the observation of the execution time, it is prudent (see, e.g., [21]) to relax the notion of noninterference from exact equality in timing observations to a parameterized neighborhood. For a given $\varepsilon > 0$, a program $\mathcal{P}$ satisfies *ε-approximate noninterference* if:

$$\forall \mathbf{y} \in \mathbb{R}^m \forall s, s' \in \Sigma \text{ we have } |\delta(s, \mathbf{y}) - \delta(s', \mathbf{y})| \leq \varepsilon. \quad (2)$$

We adapt the notion of ε-approximate noninterference in our setting of functional observations by generalizing previous notions of noninterference. We say that a program satisfies *functional ε-approximate noninterference* if

$$\forall s, s' \in \Sigma \text{ we have } d_{i,p}(\delta(s), \delta(s')) \leq \varepsilon, \quad (3)$$

where $d_{i,p}$ is a distance function over functional observations defined earlier. For example, the distance $d_{0,\infty}$ in the definition (3) recovers the definition (2). For the rest of the paper, we assume a fixed distance function d over functions.

D. Quantifying Information Leakage

The notion of noninterference requires that the attacker should deduce nothing about the secret inputs from observing the execution times for various public inputs. However, one can argue that achieving noninterference is neither possible nor desirable, because oftentimes, programs need to reveal information that depends on the secret inputs. We therefore need

a notion of information leakage. The number of distinguishable classes in timing observations often provide a realistic measure to evaluate the strength of information leaks. For example, the min-entropy measure [9] quantifies the amounts of information leaks based on the number of distinguishable observations. Our data-driven approach with functional clustering algorithms provides a lower-bound on the classes of observations.

V. Data-Driven Discovery and Explanations

The space of program inputs are often too large (potentially infinite) to exhaustively explore even for medium-sized programs. This necessitates a data-driven approach for discovery and explanation of functional side channels. In the proposed approach, an analyst uses fuzzing techniques, previously reported issues, or domain knowledge to obtain a set of secret and public inputs. In particular, an extension of gray-box evolutionary search algorithms can be used to generate interesting inputs for functional side channel analysis. Our technique then exploits functional patterns in the given inputs and applies functional data clusterings to discover functional side channels. To explain the discovered side channels, our tool instruments the programs to print information about auxiliary features (e.g., the number of times a method called or basic block executed) and apply classification inferences to localize code regions cause the side channel leaks. To summarize: given a set of program input traces, the key computational problems are a) to cluster traces exhibiting distinguishable timing behaviors and b) to explain these differences by exploiting richer information based on the auxiliary features.

Hyper-trace Learning. Let $Z{=}\{z_1, \ldots, z_r\}$ be the set of auxiliary features. An *execution trace* of a program $\mathcal{P}$ is a tuple

$$(\mathbf{x}, \mathbf{y}, \mathbf{z}, t) \in \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R}^r \times \mathbb{R},$$

wherein $\mathbf{x} \in \Sigma \subset \mathbb{R}^n$ is a value to the secret inputs, $\mathbf{y} \in \mathbb{R}^m$ is a value to the public inputs, $\mathbf{z} \in \mathbb{R}^r$ are the valuations to the auxiliary features, and $t \in \mathbb{R}_{\geq 0}$ is the execution time. We assume the valuations of the auxiliary features deterministically depend only on the secret and public inputs. To keep the execution time unaffected from the instrumentations, we estimate the execution time on the un-instrumented programs. Let $\mathcal{T}$ be a set of execution traces.

As our main objective is to explain the differences on functional observations due to differences on secret and auxiliary features, we rearrange the raw execution traces $\mathcal{T}$ to functional traces $\mathcal{H}$ by combining traces with common values of the secret inputs. Functional traces $\mathcal{H}$ are hyper-traces—as they summarize multiple program executions—that model auxiliary and timing values as a function of public inputs. A *hyper-trace* τ is a tuple

$$\big(\mathbf{x}, (\mathcal{A}_i(\mathbf{x}))_{i=1}^r, f_T(\mathbf{x})\big) \in \mathbb{R}^n \times ([\mathbb{R}^m \to \mathbb{R}])^r \times [\mathbb{R}^m \to \mathbb{R}],$$

wherein $\mathbf{x}$ is a value to the secret input, $\mathcal{A}_i$ and f_T are functions modeling values of auxiliary features and execution time, respectively, as a function of public inputs for secret $\mathbf{x}$. Computation of hyper-traces from a set of raw-traces is achieved by turning the discrete vectors of observations (for auxiliary variables as well as execution time) into smooth functions represented as linear combinations of appropriate basis functions (e.g. B-spline basis system, Fourier basis functions, and polynomial bases) [22]. We primarily use B-splines.

Side-Channel Discovery. Given a set $\mathcal{H}$ of hyper-traces, $\mathcal{H} = \{(\mathbf{x}_j, (\mathcal{A}_i(\mathbf{x}_j))_{i=1}^r, f_T(\mathbf{x}_j))\}_{j=1}^N$, we use functional data clustering over $C = \{f_T(\mathbf{x}_j)\}_{j=1}^N$ to detect different classes of observations such that hyper-traces within a cluster are ϵ-close according to the distance $d_{i,p}$.

Functional clustering approaches [5] can be broadly classified into non-parametric and model-based approaches. Our tool uses a non-parametric functional clustering and implements two algorithms to cluster indistinguishable observations. These algorithms—described in Section VI-A—take the timing observations set C, an upper bound K on the number of clusters, a distance function $d_{i,p}$, and the tolerance $\varepsilon > 0$ as inputs, and returns the "centroids" of observational functions $\mathcal{F} = \{f_1, f_2, \ldots, f_k\}$ for $k \leq K$. Our algorithm guarantees that each centroid $f_\kappa \in \mathcal{F}$ $(1 \leq \kappa \leq k)$ represents the timing functions for the set of secret values Σ_κ such that $\mathbf{x}, \mathbf{x}' \in \Sigma_\kappa$ if and only if $d_{i,p}(f_T(\mathbf{x}), f_T(\mathbf{x}')) \leq \varepsilon$.

Side-Channel Explanation. A *(hyper) trace discriminant* is defined as a disjoint partitioning of the auxiliary variables (functional) spaces along with a functional observation for each partition. Formally, a trace discriminant $\Psi = (\mathcal{F}, \Phi)$ is a set of functional observations $\mathcal{F} = \{f_1, f_2, \ldots, f_k\}$—where each $f_\kappa : \mathbb{R}^m \to \mathbb{R}_{\geq 0}$ models the execution time as a function of the public input—and a partition $\Phi = \langle \phi_1, \phi_2, \ldots, \phi_k \rangle$ where each

$$\phi_\kappa \ : \ [\mathbb{R}^m \to \mathbb{R}]^r \to \{\texttt{True}, \texttt{False}\}$$

is a predicate over the functional auxiliary features. We define $\texttt{size}(\Psi)$ as the number of functions in the discriminant Ψ, i.e. $\texttt{size}(\Psi) = |\mathcal{F}| = k$.

Given a hyper-trace $\tau{=}(x, (\mathcal{A}_i)_{i=1}^r, f_T)$ and discriminant $\Psi{=}(\mathcal{F}, \Phi)$, we define the prediction error $e(\tau, \Psi)$ as $d_{0,2}(f_T, f_\kappa)$ where $1{\leq}\kappa{\leq}k$ is the index of the unique value in Ψ such that $(x, (\mathcal{A}_i)_{i=1}^r) \models \phi_\kappa$ i.e. the predicate ϕ_κ evaluates to true for the valuation of secret value x and the functional auxiliary features $(\mathcal{A}_i)_{i=1}^r$. This evaluation triggers the functional observation f_κ. Given a set of hyper-traces $\mathcal{H} = \{\tau(\mathbf{x}_j)\}_{j=1}^N$, and a discriminant Ψ, we define the fitness of the discriminant as the mean of prediction errors:

$$\mu(\mathcal{H}, \Psi) = \frac{1}{N} \sum_{i=1}^{N} e(\tau(\mathbf{x}_j), \Psi).$$

Definition V.1 (Discriminant Learning Problem)**.** *Given a set of hyper traces $\mathcal{H}$, a bound on the size of the discriminant $K \in \mathbb{N}$, a bound on the error $\delta \in \mathbb{R}$, the* discriminant learning problem *is to find a model $\Psi = (\mathcal{F}, \Phi)$ with $\texttt{size}(\Psi) \leq K$ and prediction error $\mu(\mathcal{H}, \Psi) \leq \delta$.*

It follows from Theorem 1 in [23] that the discriminant learning problem is NP-HARD. For this reason, we propose a practical solution to the discriminant learning problem by exploiting functional data clustering and decision tree learning.

For learning the discriminant model, we adapt a decision tree learning algorithm by converting various functional data-values into categorical variables. For the r auxiliary features

Algorithm 1: $(\mathcal{F}, \Phi)$ = FUCHSIA$(\mathcal{P}, \mathcal{P}', \prec_Y, K, d_{i,p}, \varepsilon)$

Input: Program $\mathcal{P}$, the instrumentation $\mathcal{P}'$, order of public values $\prec_Y$, cluster bound K, distance $d_{i,p}$, and tolerance ε.
Output: Discovered timing observations as functional clusters and their explanations as decision tree models.

$\Pi, \Sigma =$ FUNCFUZZ$(\mathcal{P}, \prec_Y)$ ▷ Obtain secret and public sets by fuzzing $\mathcal{P}$ given an order over public input domain $\prec_Y$.
$F =$ EXECTIME$(\mathcal{P}, \Pi, \Sigma)$ ▷ Obtain timing functions $F = \{f_T(s_i)\}_{i=1}^{n}$ by executing $\mathcal{P}$ on the set Π for each secret s_i.
$\mathcal{A} =$ EXECAUX$(\mathcal{P}', \Pi, \Sigma)$ ▷ Obtain feature set $\mathcal{A}$ by executing $\mathcal{P}'$ similar to the execution of EXECTIME.
$\mathcal{F} =$ FDCLUSTERING$(T, K, d_{i,p}, \varepsilon)$ ▷ Obtain functional clusters $\mathcal{F} = \langle f_1, f_2, \ldots, f_k \rangle$ over F given K, $d_{i,p}$, and ϵ.
$\Phi =$ DISCLEARNING$(\mathcal{A}, \mathcal{F})$ ▷ Obtain discriminant $\Phi = \langle \phi_1, \phi_2, \ldots, \phi_k \rangle$ given the set $\mathcal{A}$ and functional clusters $\mathcal{F}$.
return $\mathcal{F}$, Φ

evaluated for a secret $\mathbf{x} \in \Sigma$, $(\mathbf{x}, (\mathcal{A}_i(\mathbf{x}))_{i=1}^{r})$, our algorithm clusters each auxiliary feature into k groups by employing functional data clustering [5]. Let $(\mathbf{x}, (L_i(\mathbf{x}))_{i=1}^{r})$ shows secret value $\mathbf{x}$ and categorical feature variable $L_i = \{\ell_i^1, \ell_i^2, \ldots, \ell_i^k\}$ for $i = 1, \ldots, r$. Given the set of traces $(\mathbf{x}_j, (L_i(\mathbf{x}_j))_{i=1}^{r}, f_\kappa)$ with r categorical auxiliary features and the timing function labeled with cluster color κ ($1 \leq \kappa \leq k$), the decision tree inference learns hyper-trace discriminants efficiently.

Overall Algorithm. The workflow of FUCHSIA is given in Algorithm 1. We provide a brief overview of each component of FUCHSIA here and describe the details of implementations in the next section. Given the program $\mathcal{P}$ with the secret and public inputs where $\prec_Y$ defines an arbitrary order over public input domains, the procedure FUNCFUZZ employs a gray-box evolutionary search algorithm to generate public and secret input values. The procedure EXECTIME models timing functions over the public input set for each secret value on the program $\mathcal{P}$. The procedure EXECAUX produces the auxiliary features (method calls and basic-block invocations) by executing the same inputs as EXECTIME on the instrumented program $\mathcal{P}'$. Furthermore, the procedure EXECAUX models the feature evaluations as functional objects over public inputs. Given an upper bound on the number of clusters K, the distance function $d_{i,p}$, and the tolerance ϵ, FDCLUSTERING applies a functional data clustering algorithm to find classes of observations $\mathcal{F} = \langle f_1, f_2, \ldots, f_k \rangle$. Each cluster f_i includes a set of timing functions (corresponds to a set of secret values). The procedure DISCLEARNING learns a set of discriminant predicates $\langle \phi_1, \phi_2, \ldots, \phi_k \rangle$, one predicate for each cluster defined over auxiliary features, using decision tree inferences.

VI. IMPLEMENTATION DETAILS

A. Implementations of components in FUCHSIA

FUNCFUZZ component. We implement fuzzing for our functional side channel discovery using an extension of AFL [3] and Kelinci [4] similar to DifFuzz [7]. The cost notion is the number of bytecode executed for a given secret and public pair. In our fuzzing framework, we generate multiple public values for each given secret value. Then, we model the cost of each secret value as a simple linear function (for the efficiency of fuzzer) from the domain of public values to the cost of execution. This helps exploit simple functional dependencies of response time (abstracted in the number of bytecode executed) on public inputs. During fuzzing, we record the linear cost functions obtained for different secret values. The fuzzing engine receives small rewards when the linear model of a secret value has changed and larger rewards when a new linear model found that is different than any other models (in the same public input domain) observed so far. Notice that these rewards are in addition to the internal rewards in AFL such as when it finds a new path in the program.

EXECTIME component. For each secret value, we have a vector of execution times over public inputs. We use functional data analysis tools [13] to create B-spline basis and fit functions to the vector of timing observations. The bases are a set of linear functions that are independent of one another. Given a known basis, B-spline models can approximate any arbitrary functions (see [22] for more details). The output of this step is a set of timing functions each for a distinct secret.

EXECAUX component. We use Javassist [15] to instrument any methods in a given package. The instrumented program $\mathcal{P}'$ provides us with the feature set Z that is method and basic block calls. For each secret value, we have a vector of the number of calls to the basic blocks and methods over the public inputs. We generally fit B-spline over the valuations of each auxiliary feature $z \in Z$, but we allow for simpler functions such as polynomials. The result of this step is the set $\mathcal{A}$ that defines functional values of auxiliary feature $z \in Z$ in the domain of public inputs.

FDCLUSTERING component. Given an upper-bound K on the number of clusters and the distance norm d with the tolerance ε, we implement FDCLUSTERING to discover k clusters ($k \leq K$). This clustering is an instantiation of non-parametric functional data clustering [6]. We use two algorithms: hierarchal [24] and constrained K-means [25].

Preparation for clustering. The input for the clustering is the timing functions from EXECTIME component. We use the distance function $d_{i,p}$ to obtain the distance matrix D. The distance matrix quantifies the distance between any timing functions. We specify cannot-link constraints over the matrix D. Cannot-link constraints disallow two functions that are more than ε far to be in the same cluster.

Constrained K-means clustering. Given the upper bound K, constrained K-means algorithm [26] obtains k clusters in each iteration (k = 1 in the first iteration). If the algorithm could not find k clusters ($k \leq K$) with the cannot-link constraints, it increases k to $k + 1$ and runs the next iteration. Otherwise, it returns the cluster object $\mathcal{F} = \langle f_1, f_2, \ldots, f_k \rangle$. The constrained K-means with cannot-link constraints is known to be computationally intractable [27].

Hierarchical clustering. The clustering algorithm with complete link method [28] obtains k clusters ($k \leq K$). In each iteration (k = 1 in the first iteration), it applies the hierarchal clustering, and then checks the cannot-link constraints to

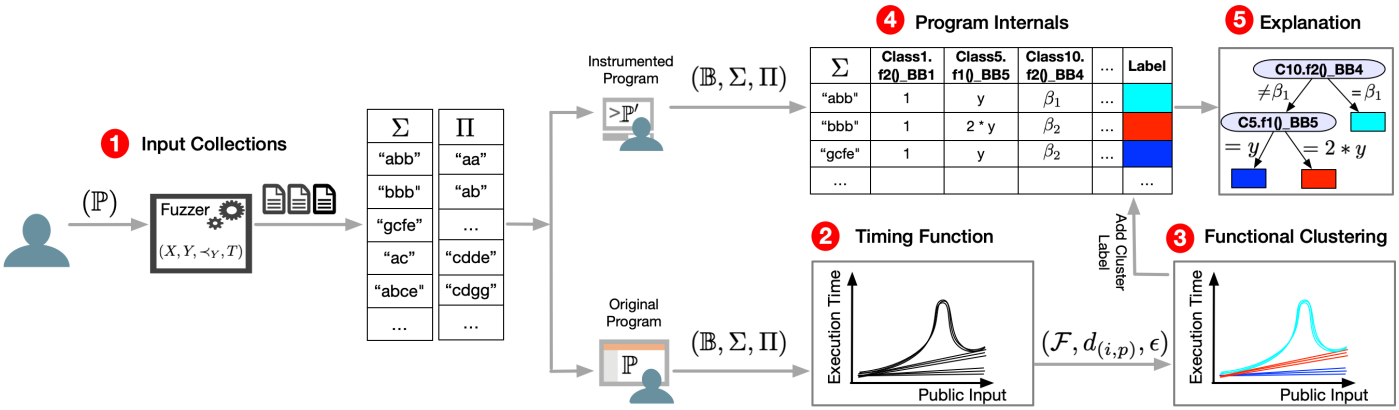


Fig. 5: FUCHSIA framework. (1) The defender feeds the program $\mathcal{P}$ to the fuzzing engine and generates a set of secret (Σ) and public (Π) inputs. (2) The defender specifies the basis function $\mathbb{B}$ (such as B-Spline basis) and enables FUCHSIA to generate timing functions $\mathcal{F}$. (3) Given the functions, the defender specifies the tolerance ε and the distance norm (such as L_1-norm), and FUCHSIA identifies the clusters in timing functions. (4) On the instrumented version $\mathcal{P}'$, FUCHSIA uses the same basis $\mathbb{B}$ and models the calls to each basic block with functions. (5) Given the cluster label and the basic block evaluations, FUCHSIA applies decision trees to explain side channels with program internal properties.

make sure that all the functions in the cannot-link set are in different clusters. If the condition is not satisfied, it increases k to $k+1$ and runs the next iteration. Otherwise, it returns $\mathcal{F} = \langle f_1, f_2, \ldots, f_k \rangle$. Hierarchical clustering is agnostic to the constraints, and the constraints are checked after the clustering.

Point-wise clustering. We use the definition of well-establish ϵ-approximate noninterference in [21], [7] for point-wise clustering. For every public input value, we form cannot-link constrains and apply one of the clustering algorithms with the ∞-norm and the tolerance ε. Finally, we choose the largest number of clusters among all values of the public inputs.

DISCLEARNING component. Using the auxiliary variables $\mathcal{A}$ as features and the functional clusters $\mathcal{F}$ as labels, the problem of learning discriminant models becomes a standard classification problem. The white-box decision tree model explains what auxiliary features are contributing to different clusters. We use the CART decision tree algorithm [29].

FUCHSIA framework. Figure 5 shows the steps of FUCHSIA for a defender to discover and explain timing side channels.

(i) The user (defender) starts interacting with FUCHSIA by feeding the program or library $\mathcal{P}$ to the fuzzing engine. This involves modifying the fuzzing driver to call the main method in program $\mathcal{P}$ with secret input variables X, public input variables Y, and an order on the public input $\prec_Y$. FUCHSIA supports all variable types and provides various options for ordering the public input variables including the size (default), the lexicographic order, and the number of set bits. The user can optionally tune the parameter determining the number of public values to be generated per each secret value in the fuzzing driver (the default value for this parameter is 3).
The user then invokes the fuzzer and has an option of stopping it either after a pre-specified timeout T (default is 2 hours), or when a desired number of inputs is generated. After the fuzzing, the user gathers the set of secret (Σ) and public (Π) inputs. Optionally, the user can specify any other desired set of inputs with unexpected behaviors.

(ii) In the next step, FUCHSIA identifies timing functions for each secret input generated in the first step. The defender has the option to choose the basis $\mathbb{B}$ for timing functions, such as B-splines (default) or polynomials. For each secret input, FUCHSIA runs the program $\mathcal{P}$ on the set of public inputs Π, measures the response times, and fits the basis to generate the timing function. The defender obtains the set of timing functions $\mathcal{F}$, one for each secret value. Optionally, the defender may use the number of byte-code executed instead of the actual response time.

(iii) Next, FUCHSIA identifies natural clusters in this set of timing functions $\mathcal{F}$. To aid this, the defender provides the distance norm $d_{i,p}$ and a tolerance ε, and FUCHSIA returns the cluster label for each timing function. The implemented options for the $d_{i,p}$-norm include L_1-norm (p = 1, default), L_∞-norm (p = ∞), and L_2-norm (p = 2) for the timing functions ($i = 0$, default) and their first derivatives ($i = 1$).
The parameter ε (with the default value of 0.1) can be fine-tuned based on the noises present the timing observations using the following procedure: a) select a secret value randomly; b) run the program (with that secret value) multiple times on the set of public values; c) create several timing functions and employ the clustering algorithm; d) search for the smallest value of tolerance ε such that the algorithm returns one cluster. This sampling procedure can be repeated multiple times (with different secrets) to get more precise estimates. The accuracy of decision tree is another key criterion to base the tuning of the ε parameter and choose values leading to accurate trees.

(iv) The fourth step is to generate program internal traces for the inputs reported in the first step. FUCHSIA allows the user to specify *features* over program

internals—such as basic blocks traversed and set of methods invoked—to base the explanation of the timing side channels. FUCHSIA runs the same set of secret and public inputs on the instrumented program $\mathcal{P}'$ to collect data about these features. This results in a rich summary of the program traces expressed as the values of these features. FUCHSIA uses the same basis $\mathbb{B}$ (default) and model the number of calls to each basic block with functions.

(v) In the last step, given the basic block evaluations and cluster label for each secret value, FUCHSIA uses the decision tree models to localize code regions that contribute to the creation of timing side channels. This step does not require parameters from the defender.

B. Environment Setup

All timing measurements in EXECTIME of Algorithm 1 are conducted on an Intel NUC5i5RYH [30]. We run each experiment 10 times and use the mean for the analysis. All other components are conducted on an Intel i5-2.7 GHz with 8 GB RAM. The FUCHSIA includes almost 2,000 lines of code. The functional analysis and clustering are implemented in R using functional data analysis package [31] and hierarchal clustering package [28]. The fuzzing and instrumentations are implemented in Java using AFL [3], Kelinci [4], and Javassist [32]. The decision tree learning algorithm is implemented in python using scikit-learn library [33].

C. Micro-benchmarks

We first compare the two clustering algorithms from Section VI-A. Then, we examine the scalability of different components in FUCHSIA. Finally, we study and compare the results of FUCHSIA versus DifFuzz [7].

Programs. Two programs `Zigzag` and `processBid` are shown in Figure 7. The applications `Guess_Secret_1` [34] and `Guess_Secret_2` [35] (shown in Figure 7) take the secret and public as the inputs and execute different sleep commands depending on their values. `PWCheck_unsafe` is a password checking example taken from [7]. Six versions of branch and loop are considered, with one depicted in Figure 7. Depending on the secret value, the program does computations with four types of complexities: O(log(N)), O(N), O(N.log(N)), and $O(N^2)$ where N is the public input. Each branch and loop program has all four loop complexities with different constant factors such as O(log(N)) and O(2.log(N)).

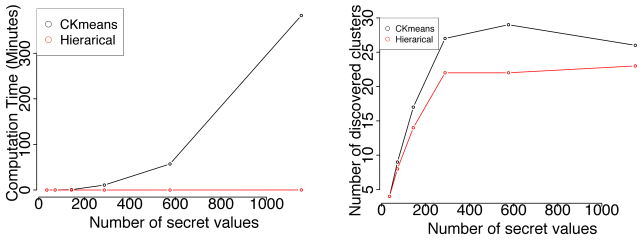


Fig. 6: (a) Computation time. Hierarchical clustering is scalable better than constrained K-means. (b) Number of clusters. Hierarchical clustering discovers a fewer number of clusters.

Input Generations. To study the micro-benchmark programs, we generate inputs using our fuzzing technique. We run the fuzzing for 30 minutes on each program and use the generated inputs for the rest of analysis.

Clustering Parameters. We use both the functional data clusterings (constrained K-means and hierarchical) as well as point-wise clustering. For the point-wise clustering, the distance is based on ∞-norm. For functional clusterings, we consider the L_1-norm distance (d_1) with the tolerance ε_1.

Clustering Comparison. Figure 6 shows the comparison between the hierarchical and constrained K-means algorithms (Section VI-A) using `Branch_and_Loop_6` where the number of secrets varies from 32 to 1,024 all with 1,000 public values. It shows that the constrained K-means is computationally expensive, while the hierarchical clustering is much more scalable (up to $400\times$). Besides, the constrained K-means discovers more clusters than the hierarchical one. Note that the clusters discovered by both algorithms are valid, and we prefer the one with the fewer number of clusters! We use the hierarchical algorithm for the rest of this paper.

Scalability. We examine the scalability of components in FUCHSIA for fitting functions, finding clusters, and learning decision trees. We observe that FUCHSIA can handle more than 250 timing functions each defined over more than 2,000 public values in less than 30 seconds. The computation time grows in the quadratic factor with respect to the growth of the number of secret values and public values. Learning the decision tree model includes both fitting functions over the auxiliary features and using CART algorithms in the functional domain. We observe that this procedure is scalable and takes less than one minute in the worst case.

DifFuzz Approach. DifFuzz is a recent side-channel detection technique that outperforms other state-of-the-art techniques [7]. The approach extends AFL [3] and Kelinci [4] fuzzers to detect side channels. The goal of DifFuzz is to maximize the following objective: $\delta = |c(p, s_1) - c(p, s_2)|$, that is, to find two distinct secret values s_1, s_2 and a public value p that give the maximum cost (c) difference. Because the cost function (c) is the number of executed bytecodes, we use BigInteger manipulations equivalent to sleep commands in micro-benchmark programs. Note that the objective function is based on ϵ-approximate noninterference where the goal is to maximize the point-wise cost differences between two secrets.

DifFuzz versus FUCHSIA. Table I shows the results of applying FUCHSIA and DifFuzz on the set of micro-benchmark programs. We generate inputs for both approaches in 30 mins. We analyze the input generated from these two approaches in two criteria: 1) whether they deem the benchmark safe? 2) how many classes of observations do they find? Based on the results in [7], the minimum value of cost difference δ for unsafe variants is 8. However, in this study, a program is safe if $\delta \leq 1$ that allows DifFuzz to deem the application unsafe for smaller cost differences. In the same way, we set the tolerance parameter based on L_1 norm distance in FUCHSIA to be 1.

We highlighted differences between DifFuzz and FUCHSIA in Table I. First, DifFuzz reports the max. cost difference in `Zigzag` application as 1, so the program is safe. This is largely due to the point-wise noninterference definition in DifFuzz. The definitions with the L_1-norm over the shapes

Fig. 7: Sample programs used in Micro-benchmark analysis.

① Zigzag ② processBid

```
Zigzag(int secret, int low) {
  if(secret % 2 == 0){
    if (low % 2 == 0) Thread.sleep(3);
    else Thread.sleep(1);}
  else Thread.sleep(2);}

processBid(int sec, int offer){
  if (offer < secret) return false;
  else { recordBid(offer); return true;}}
```

③ Branch_loop_1

```
Branch_loop_1(int secret, int N) {
  if (secret<100) for(int i=N; i>0; i /= 2)  Thread.sleep(1);
  else if (secret<195) for(int i=0;i<N; i++) Thread.sleep(1);
  else if (secret < 290) for (int i = 0; i < N; i++) {
      for (int j = N; j > 0; j /= 2) Thread.sleep(1);
    }
  else if (secret < 400) for(int i = 0; i < N; i++) {
      for(int j = 0; j < N; j++) Thread.sleep(1);
  }}
```

④ Guess_Sec_2

```
Guess_Sec_2(int secret,int low,int t) {
  if(low <= secret) {
    if(t == 1) Thread.sleep(1);
    else if (t == 2) Thread.sleep(10);
    else Thread.sleep(1000);}
  else {
    if (t == 1) Thread.sleep(1);
    else if(t == 2) Thread.sleep(100);
    else Thread.sleep(1000);}}
```

TABLE I: Micro-benchmark results for FUCHSIA and DifFuzz [7]. Legends: #**R**: no. of methods, #$\mathbf{S}_F$: no. of secret values (FUCHSIA), #$\mathbf{P}_F$: no. of public values (FUCHSIA), ε_1: tolerance for L_1-norm functional clustering (FUCHSIA), #$\mathbf{K}_F$: no. of clusters (FUCHSIA), **Safe**$_F$: Yes, if there is only 1 cluster (FUCHSIA), #$\mathbf{S}_D$: no. of secret values (DifFuzz), #$\mathbf{P}_D$: no. of public values (DifFuzz), δ: max. cost difference (DifFuzz), **Safe**$_D$: Yes, if $\delta \leq 1$ (DifFuzz), #$\mathbf{K}_D$: no. of clusters (DifFuzz).

		FUCHSIA					DifFuzz [7]				
Benchmark	#**R**	#$\mathbf{S}_F$	#$\mathbf{P}_F$	ε_1	**Safe**$_F$	#$\mathbf{K}_F$	#$\mathbf{S}_D$	#$\mathbf{P}_D$	**Max.** δ	**Safe**$_D$	#$\mathbf{K}_D$
Zigzag	13	70	6,912	1	No	2	3,007	1,532	1	Yes	1
Guess_Secret_1	10	110	6,649	1	No	105	9,672	4,797	2	No	2
Guess_Secret_2	5	72	7,414	1	No	7	6,480	3,476	0	Yes	1
processBid	3	116	8,100	1	No	112	2,282	1,170	4	No	2
pwcheck_unsafe	3	118	9,560	1	No	115	15,290	7,660	47	No	16
Branch_and_Loop_1	4	179	1,761	1	No	4	8,303	4,477	30,404	No	4
Branch_and_Loop_2	8	226	2,111	1	No	5	9,003	4,524	30,404	No	5
Branch_and_Loop_3	16	224	2,101	1	No	7	4,419	2,556	30,405	No	6
Branch_and_Loop_4	32	229	2,121	1	No	9	8,612	4,656	30,405	No	6
Branch_and_Loop_5	64	238	2,213	1	No	19	10,523	5,337	30,405	No	7
Branch_and_Loop_6	128	255	2,200	1	No	24	7,539	3,869	30,405	No	5

can easily show higher costs and deem the application unsafe. Second, we apply the point-wise and functional clusterings for inputs generated by DifFuzz and FUCHSIA, respectively. We observe that DifFuzz finds fewer clusters compared to FUCHSIA. There are mainly two reasons for these differences. The first factor is due to the point-wise definitions in finding classes of side channels as illustrated in Section II-B. In `Guess_1` program, for each distinct secret value, there is a unique public value where the execution time jumps from one cost to another. These are captured by the functional clustering where there is an almost equal number of secrets and clusters. The second one is due to the objective function of DifFuzz that tries to find two secret values (with the same public value) such that the cost differences between them are maximized. FUCHSIA, on the other hand, tries to find as many functional clusters as possible. This factor is the main reason for the differences in `Branch_and_loop` applications.

VII. CASE STUDIES

Table II summarizes 10 Java applications used as case studies. We consider L_1-norm distance between timing functions ($\varepsilon_{0,1}$) and their first derivatives ($\varepsilon_{1,1}$). The main research questions are "Do functional clustering and decision tree learning (a) scale well and (b) provide useful information about leaks?"

A) Regex. Regex's case study was described in Section III. To answer the research question: *Usefulness:* The decision tree pinpoints a location in the regex package that leaks the value of secret patterns. *Scalability:* The overall computation time of clustering and decision tree learning is about 65 mins.

B) Jetty. We analyze the `util.security` package of Eclipse Jetty web server. The package has a `Credential` class which had a timing side channel. This vulnerability was analyzed in [21] and fixed initially in [11]. Then, the developers noticed that the implementation in [11] can still leak information and fixed this issue with a new implementation in [36]. We consider this new implementation shown in Figure 9 and apply FUCHSIA to check its security. The final fix was done a few months later [37], but before we reported our finding to the developers.

Inputs. The secret input is the password stored at the server, and the public input is the guess. The defender starts by choosing a finite set of secret and public values from the fuzzer. The defender obtains 800 different secret passwords and 635 different guesses from the fuzzer. The lengths of passwords are at most 20 characters.

Side Channel Discovery. For each secret value, FUCHSIA varies 635 different guesses and measures the execution time of Jetty. Then, FUCHSIA models the running time of 800 secret values with B-spline basis. The next step is to find out how these functions are related based on their functional distances. Given the L_1-norm as the distance function and the tolerance $\varepsilon = 0.1$, FUCHSIA uses the clustering algorithm and returns 20 classes of observations as shown in Figure 8 (a). The existence of 20 distinct classes of observations indicates the presence of a functional side channel in the Jetty package.

Side Channel Explanation. Now, the defender wants to know what properties of program internals leak through the timing side channels. FUCHSIA uses the instrumented Jetty and obtains 56 internal features such as method calls and basic block invocations. Each secret value has the functional evaluation of 56 internal features over the public inputs as

TABLE II: Case Studies. Legends similar to Table I, except, **#M** the number of methods in applications, $\varepsilon_{0,1}$: tolerance for L_1-norm of the timing model, $\varepsilon_{1,1}$: tolerance for L_1-norm of the first derivative of the timing model, **A**: accuracy of the tree model, **H**: height of the tree, **#L**: number of leaf nodes in the tree, **T**: computation time for decision tree learning (s).

Benchmark	#M	#R	#S	#P	$\varepsilon_{0,1}$	$\#\mathbf{K}_{0,1}$	$\mathbf{T}_{0,1}$	$\varepsilon_{1,1}$	$\#\mathbf{K}_{1,1}$	$\mathbf{T}_{1,1}$	A	H	#L	T
Regex	620	203	1,154	6,365	2e-1	162	1,801	2e-1	49	4,812	89.7%	14	120	2,084.0
Jetty	63	56	800	635	1e-1	20	49.7	1e-2	15	82.6	99.4%	12	20	52.1
iControl (SOAP)	41,541	127	342	1,164	1e-1	33	19.7	1e-1	19	43.2	98.2%	10	9	10.5
Javax (crypto)	612	56	1,533	1,045	1e-1	54	174.2	1e-1	32	253.0	88.6%	6	36	7.2
GabFeed	573	43	1,105	65	1e-1	34	58.5	1e-2	34	70.5	99.6%	31	34	41.7
Stegosaurus	237	96	512	60	2e-1	5	3.6	1e-1	3	3.6	100.0%	4	5	12.6
SnapBuddy	3,071	65	477	14	2e-1	13	2.8	2e-1	8	3.0	96.2%	14	13	3.1
ShareValue	13	7	164	41	6e-2	29	0.7	1e-2	14	0.7	99.3%	17	29	3.4
MST(Kruskal)	5	6	120	40	3e-1	20	0.4	3e-1	5	0.4	80%	7	20	3.0
Collab	185	53	176	11	1e-2	1	0.3	1e-2	1	0.3	N/A	N/A	N/A	N/A

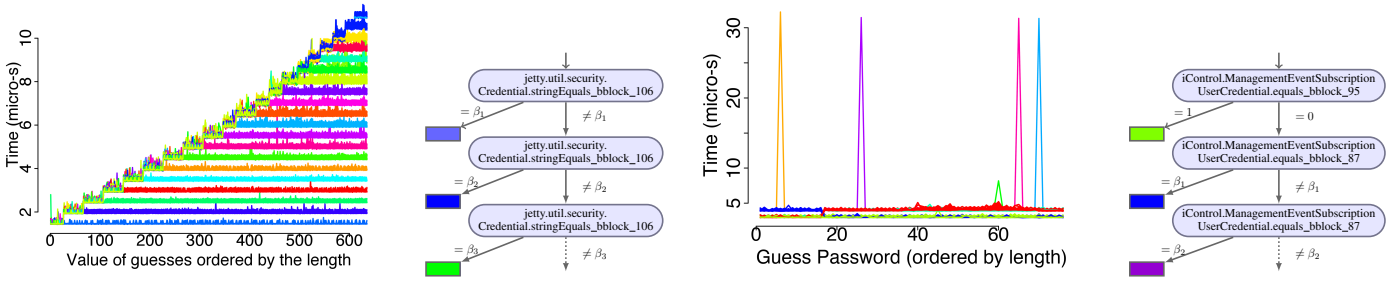


Fig. 8: (a) 800 Jetty timing functions are clustered into 20 groups using L_1-norm. This indicates potential timing leaks in Jetty. What properties are leaking? (b) Jetty decision tree. The number of calls to the basic block at line 106 of stringEquals (shown in Figure 9) discriminates different clusters. The code region shows the length of secret passwords is leaking. (c) (parts of) 342 iControl timing functions are clustered into 9 groups. (d) (parts of) iControl decision tree model. It pinpoints basic blocks in `UserCredential.equals` method. The code region indicates that the whole secret password can be compromised with timing side channel attacks.

Fig. 9: String equality in Eclipse Jetty (`s1` secret, `s2` public).

stringEquals

```
boolean stringEquals(String s1, String s2) {
  if (s1 == s2) return true;
  if (s1 == null || s2 == null) return false;
  boolean result = true;
  int l1 = s1.length(), l2 = s2.length();
  if (l1 != l2) result = false;
  int l = Math.min(l1, l2);
  for (int i = 0; i < l; ++i){     (line.106)
    result &= (s1.charAt(i) == s2.charAt(i));}
  return result;
  }
```

well as a label from the clustering. Next, FUCHSIA uses the decision tree inferences to localize code regions that contribute to different observations. Figure 8 (b) shows (parts of) the decision tree model learned for Jetty. Using this model, the defender realizes that different calls to a basic block in `Credential.StringEquals()` method are what distinguishes the clusters. This basic block represents the loop body of the `for` statement in the method shown in Figure 9. For instance, the green cluster (third from the bottom of the center diagram, the bottom of the right diagram) corresponds to the case where `stringEquals_bblock_106` is executed according to β_3 function. Note that edge values are B-spline functions over public values, and the max value for β_i function is i. For the example of β_3, the max value is 3, and if the basic block for a secret value is called at most three times, it belongs to the green cluster. Using the decision tree model and the relevant code, the defender realizes that the minimum of the lengths (the secret and the guess) is leaking through the calls to `stringEquals_bblock_106`.

Usefulness: The decision tree pinpoints a location in Jetty that leaks the length of secrets. *Scalability:* The overall computation time is about 2 mins.

C) iControl (SOAP). iControl (SOAP)[2] is an open source API that uses SOAP/XML to establish communications between dissimilar systems. The library has 41,541 methods. One key confidentiality-related functionality of the library is to store credentials of various users and to validate their credentials against a given guess. The defender's goal is to find out whether there exist timing side channels in the library, and if so, identify the code regions potentially responsible for creating the side channels.

Inputs. The natural candidate for the secret input in this application is the stored credential at the server, while the public input is a given guess against a stored credential. The defender considers the lexicographic ordering over public

[2]https://clouddocs.f5.com/api/icontrol-soap/

inputs to generate functional data. With a timeout of two hours on fuzzing, the defender obtains 342 unique secret credentials and 1,164 unique guesses. The credentials include secret passwords with the length of at most 16 characters.

Side Channel Discovery. FUCHSIA begins by varying 1,164 guesses for each secret input and uses B-spline to model 342 timing functions. With the default parameters (L_1-norm as the distance norm and $\varepsilon = 0.1$) for the clustering, FUCHSIA identifies 33 classes of observations. Figure 8 (c) shows timing functions and corresponding clusters. The presence of multiple clusters points towards the existence of timing side channels.

Side Channel Explanation. The defender specifies basic block calls as the features to be used in the explanation of the side channels. FUCHSIA runs previously identified inputs on the instrumented version of iControl and generates 127 auxiliary features (basic blocks) about the internals of iControl. Given the set of traces containing the information on these features and corresponding cluster labels, FUCHSIA uses decision tree models to present an explanation for the side channels. The decision tree is shown in Figure 8 (d). It pinpoints that different calls to the basic block at the equality check in `ManagementEventSubscriptionUserCredential` class is a potential explanation of the timing differences. Using this information and the relevant code, the defender may infer that the application uses Java string equality check to compare passwords. This leads to a password-matching style vulnerability where an attacker can obtain a prefix of secret passwords in each step of attack.

We reported this vulnerability to both F5 security team and F5 open-source community developers. The F5 security team has confirmed this vulnerability. Moreover, this explanation helped them to identify a potential vulnerability in their closed-source implementations.

D) Javax Crypto. Javax library provides the classes and interfaces for cryptographic operations in Java. The `crypto` package in the library has 612 methods and provides functionalities such as creating and modifying symmetric secret keys. We analyzed the `crypto` package of javax library[3] against timing side-channel vulnerabilities.

Inputs. The secret input is the symmetric secret key of encryption algorithms (such as "DES"), and the public input is a guess key to be compared against the secret key. During two hours of fuzzing, the defender generates 1,533 secret keys and 1,045 guess keys. The length of a key is at most 16 bytes.

Side Channel Discovery. FUCHSIA identified 1,533 timing functions using B-spline basis and returned 54 clusters, as shown in Figure 10 (a), with default parameters (L_1-norm and $\varepsilon = 0.1$). The presence of 54 classes of observations indicates the existence of timing side channels.

Side Channel Explanation. The next step is to identify the culprit code regions and understand what properties of secret keys are leaking. FUCHSIA runs the same set of secret and guess inputs over the instrumented version (to output information about the basic blocks) of the crypto library. This results in generating 56 auxiliary features about the basic block calls. Given the basic block evaluations and the cluster for each secret value, the decision tree model explains which basic blocks contribute to different timing observations. The decision tree in Figure 10 (b) shows the calls to a basic block in `spec.SecretKeySpec.equals()` method is the root cause of timing side channels:

SecretKeySpec.equals(Object obj)

```
if (this == obj)
   return true;
if (!(obj instanceof SecretKey))
   return false;
String thatAlg = ((SecretKey)obj).getAlgorithm();
if (!(thatAlg.equalsIgnoreCase(this.algorithm))) {
   ...
}
byte[] thatKey = ((SecretKey)obj).getEncoded();
return java.util.Arrays.equals(this.key, thatKey);
```

This results in calling to `util.Arrays.equals()`:

Arrays.equals(byte[] a, byte[] a2)

```
if (a==a2)  return true;
if (a==null || a2==null) return false;
int length = a.length;
if (a2.length != length) return false;
for (int i=0; i<length; i++)
   if (a[i] != a2[i]) return false;
return true;
```

This internal method for the equality check of byte arrays is vulnerable to timing side-channel attacks. The method returns as soon as there is a mismatch between two byte arrays. An attacker can exploit this vulnerability to recover secret keys.

We reported this problem to OpenJDK security team. During the discussion, we were informed that the vulnerability has since been fixed in an updated version of JDK-8 [38] (we analyzed JDK-8 project, while the fix appears in JDK-8-u project). We also analyzed the implementations in JDK-8-u project [38] with the same set of inputs from the previous analysis. During this analysis, we found out that there are 7 clusters in timing observations. This shows that the new implementation has not completely fixed the side channels. The decision tree explains that there are different calls to the basic block at line 454 in `isEqual()` method of `MessageDigest` class [39]. Looking into the source code, we observed that the length of secret byte arrays is leaking via timing side channels. Furthermore, the vulnerability applies to any functionalities in javax that compare byte arrays. We reported this vulnerability to the developers and suggested safe implementations to fix it.

E) GabFeed. GabFeed is a Java application with 573 methods implementing a chat server [21].

Inputs. The server takes users' public key and its own private key to generate a common key. The defender uses FUCHSIA to obtain 1,105 server's private keys and 65 public keys where the public keys are ordered by their number of set bits. In total, there are 71,825 test cases.

Side Channel Discovery. For each secret key, FUCHSIA varies public keys and measures the execution time to generate the common key. Next, FUCHSIA uses B-spline and creates timing functions for each secret. The next step is to find the equivalence relations over the secret input using the functional clustering. The defender provides L_1-norm and $\varepsilon_{0.1} = 0.1$ as

[3]https://hg.openjdk.java.net/jdk8/jdk8/jdk/file/687fd7c7986d/src/share/classes/javax/crypto

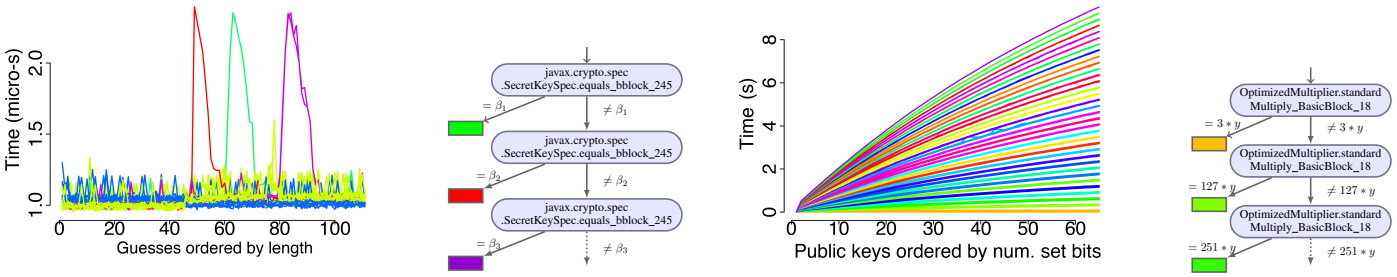


Fig. 10: (a) (parts of) 1,533 timing functions of javax Crypto are clustered into 5 groups. (b) (parts of) javax Crypto decision tree model. It localizes the basic blocks in `SecretKeySpec.equals` that can leak the entire secret key due to the use of Java internals to compare byte arrays. (c) 1,105 GabFeed timing functions are clustered into 34 groups. (d) GabFeed decision tree shows the basic block at line 18 of `standardMultiply` method is the discriminants. The code region shows the number of set bits in the secret key is leaking.

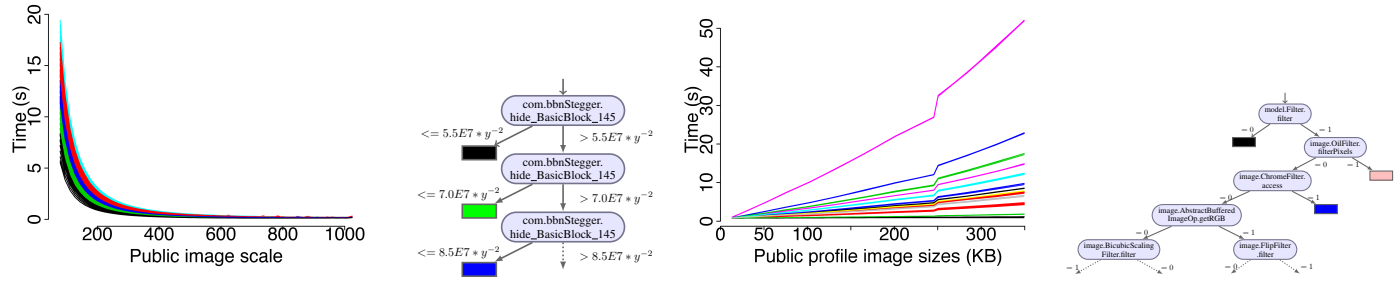


Fig. 11: (a) 512 Stegosaurus timing functions are clustered into 5 groups. (b) Stegosaurus decision tree model. It pinpoints the basic block at line 145 of the `hide` method. The code region indicates that the length of secret messages is leaking. (c) 477 timing functions of users' profiles in SnapBuddy are clustered into 13 groups. (d) SnapBuddy decision tree model: calls to photo filter functions are discriminants. The type of photo filters applied by users on their public profiles can leak their identities.

parameters to the hierarchal clustering, and FUCHSIA discovers 34 clusters as shown in Figure 10 (c). This step partitions 1,105 secret values into 34 distinguishable classes.

Side Channel Explanation. The next step is to find out what properties of secret keys are leaking. FUCHSIA runs the same secret and public inputs over the instrumented GabFeed and obtains 43 different auxiliary features. Given the basic block evaluations and the cluster for each secret value, the task is to learn what basic blocks contribute to different clusters. FUCHSIA uses the CART algorithm and produces the model in Figure 10 (d). Using this model, the defender observes that the number of basic block calls at line 18 of the `standardMultiply` method explains different clusters. The edge values in the decision tree model is a linear function of the public input with different slopes. Inspecting the source code, the basic block executes expensive shift left and add operations over BigIntegers of public keys for each set bit in the secret key. The slopes in the edge values of the decision tree model depend on the number of set bits in the secret key.

Usefulness: The decision tree explains that the calls to an expensive basic block is a linear function of public key where the slope depends on the number of set bits in secret key. GabFeed authentication algorithm leaks set bits in the secret. *Scalability:* The overall computation time is about 2 mins.

F) Stegosaurus. Stegosaurus with 237 methods is a messaging service that uses steganographic algorithms to hide secret messages [40]. The application takes the secret message with a 128-bit key and embeds the message in a random image.

Inputs. The secrets are the message and the key. The defender uses FUCHSIA to generate 512 secret messages with a size of at most 8 letters. We assume that the secret key is a fixed random value chosen by the service. The public input is an image chosen by the defender and ordered by their scales.

Side Channel Discovery. For each secret message, FUCHSIA varies the scale of images from 80×80 to 1,024×1,024. Then, it measures the execution time of the application to encode the message in different images. In total, FUCHSIA models 512 timing functions. The next step is to find the equivalence classes over these functions. The defender provides L_1-norm and $\varepsilon_{0,1} = 0.2$ as parameters to the clustering, and FUCHSIA finds 5 classes of observations shown in Figure 11 (a).

Side Channel Explanation. The next step is to find out what properties of secret messages are leaking. FUCHSIA uses the instrumented version of Stegosaurus to generate basic block calls for each execution. In total, FUCHSIA obtains 96 different basic block calls each as a function over public inputs. Given the basic block calls and the (timing) cluster of each secret value, FUCHSIA uses the CART inference to explain which of 96 basic blocks contributes to different observations. The decision tree model is shown in Figure 11 (b). The defender realizes that the number of basic block calls at line 145 of the `hide` method explains different clusters:

hide()

```
for (int pos=0; pos<=message.length(); ++pos) {
   ...
   while (pkCopy.compareTo(maxOffset) > 0) {
(l.145)   pkCopy = pkCopy.subtract(maxOffset.multiply(perf));
   }
   ...}
```

In the above code snippet, `pkCopy` is the fixed secret key, `perf` is a constant BigInteger value, and `maxOffset` is the BigInteger representation of the image scale (height×width).

Usefulness: The decision tree model shows the number of calls to the basic block at line 145 depends on the secret message length directly and the scale of the image inversely. Thus, the length of secret messages is the leaking property. *Scalability:* The overall computation time is less than 1 min.

G) SnapBuddy. SnapBuddy with 3,071 methods is a mock social network where each user has their own page with a photograph [41]. The size of profiles is a public input (observable through the generated packets), and the identity of users actively interacting with the server is a secret input.

Inputs. The defender considers the identities of 477 users currently in the network as the secret inputs and varies the size of public profiles from 13KB to 350KB.

Side Channel Discovery. FUCHSIA uses B-spline to model the profile retrieval times for each user as a function of profile sizes. FUCHSIA models 477 timing functions, one for each user. The next step is to find out the relationships between the timing functions of different users and determine if there are timing side channels. For this aim, FUCHSIA applies the clustering algorithm to identify different classes of observations. FUCHSIA discovers 13 clusters ($\varepsilon_{0,1}$=0.2) shown in Figure 11 (c). The clustering partitions timing observations for 477 users into 13 equivalence classes.

Side Channel Explanation. The next step is to find out what properties of users' public profiles are leaking. In this example, in particular, it is difficult to find out the leaking property solely based on the profile features since it is exhaustively large. Some examples are the users' locations, their names, their friends, their friends' name, their friends' location, to mention but few. This is one reason that we turn into collecting program internal features through instrumentations. The instrumentation provides 65 auxiliary features, and we model them as functions over the profile sizes. Figure 11 (d) shows (part of) the decision tree model that says users who do not apply any filter on their images follow the black cluster (the bottom cluster in Figure 11 (c)), while those who apply oilFilter on their images are assigned to the pink cluster (the top cluster in Figure 11 (c)). The decision tree shows that it is the type of photo filters applied by the users on their public profile images that are leaking. A passive attacker can use this information to reduce her uncertainty about the identity of a user whom the server downloads his/her profile, especially if some filters used by a few users in the SnapBuddy.

Usefulness: The decision tree model explains non-trivial facts about leaks. It shows that different photo filters applied by users on their profiles are leaking. The defender can use this information to debug timing differences related to the image filters. *Scalability:* The overall analysis takes less than 1 min.

H) Share Value. The application is an extension of classical share value program studied in [42], [43]. In this case, every user in the system has public and private shares. The program calculates useful statistics about shares.

Inputs. The program has 164 users each with maximum of 400 private shares. The user can have 1 to 400 public shares.

Side Channel Discovery. FUCHSIA generates private and public shares in the given range. In particular, it fixes private shares for each user. Next, FUCHSIA varies the number of public shares for each user and measure the response times to calculate the statistics for each user. Then, it fits B-spline to the execution times and applies the functional clustering that discovers 29 clusters with $\varepsilon_{0,1} = 0.06$.

Side Channel Explanation. The next step is to find out what properties of private shares are leaking by using richer information from program internals. The decision tree model shows that different intervals of calls to connect to a remote database is the root cause of the leaks. Therefore, the number of secret shares are leaking through the time required to connect a remote DB. The overall analysis takes about 4 (s).

I) Kruskal. We analyze Kruskal's algorithm [44] and its implementation in [45]. Here, we assume that a graph data structure with Kruskal's algorithm is used in a security setting where the graph nodes are public and the structure of the graph (the connection of nodes) is secret.

Inputs. The input generation for Kruskal's algorithm is based on the domain knowledge of this problem. Given that the structure of graphs is secret, the defender constructs 4,800 graphs as the following. The defender considers 120 different graph structures from the interval between a spanning tree $(n-1)$ and a complete graph $(n \times (n-1)/2)$. For each structure, the number of nodes (n) varies from 2 to 200 and the number of edges is determined based on the structure of the graph. For example, if the structure of a graph is a spanning tree, the number of edges varies from 1 to 199.

Side Channel Discovery. For each graph structure, FUCHSIA fits timing functions that are from the number of nodes to the execution time. Then, FUCHSIA applies the clustering algorithm and discovers 20 clusters with $\varepsilon_{0,1} = 0.3$. The presence of the 20 clusters indicate the possibility of information leaks about the graph structure.

Side Channel Explanation. The next step is to find out what properties of program internals are leaking and establish the facts about the leaks. We obtain program internal features and apply decision tree algorithms on the set of features for different secret values. The model shows the number of calls to the `compareTo` method distinguish different clusters. This indicates the sorting algorithm in the MST calculation that depends on the number of edges is the cause of different observations. An eavesdropper can use the side channel to guess whether the graph is a sparse graph or a dense graph. The overall analysis time takes about 4 (s).

J) Collab. Collab is a scheduling application that allows users to create a new event and modify existing ones [46]. Users can apply add, $commit$, and $search$ operations on events. An audit event is a secret, while other events are public.

Inputs. The defender considers 176 users in the system, each with either zero or one audit events. The public inputs are the operations performed on the public events of users.

Side Channel Discovery. For each user, FUCHSIA applies 1 to 11 operations randomly from the set of possible operations on their public events and measure the response times. FUCHSIA models 176 timing functions, one for each user in the system. The next step is to find out the classes of observations on these functions. FUCHSIA discovers only one cluster with a small tolerance value, and the defender concludes that no information about the audit events of users is leaking through timing side channels. The clustering algorithm takes about 1(s).

VIII. Related Work

Noninterference. Noninterference notion [18] has been widely used to enforce confidentiality in various systems [19], [20], [47]. Previous works [21], [7] extend the classical notion of noninterference with relaxed notions called ε-bounded noninterference. We adopt the well-established noninterference definition to the functional setting with various noise models.

Static Analysis for side channels. Various works [21], [48], [49], [50], [51] use static analysis for side-channel detections. Chen et al. [21] casts the noninterference property as 2-safety property [52] and uses Cartesian Hoare Logic [52] equipped with taint analysis [53] to detect side channels. These static techniques rely on the taint analysis that is computationally difficult for real-world Java applications. The work [54] reported that 78% of 461 open-source Java projects use dynamic features such as reflections that are problematic for static analysis. We use dynamic analysis that handles the reflections and scales well for the real-world applications.

Dynamic Analysis for side channels. Dynamic analysis has been used for side-channel detections [55], [7], [56], [57], [58]. We compared our technique to DifFuzz [7] in Section VI-C. Profit [56] considers a black-box model of programs and study information leaks through network traffics. It first aligns different traces of packets to identify phases in the application. Then, it extracts packet-level features such as the time differences between two packets. Finally, it uses Shannon entropy to quantify information leaks related to each feature and provide a ranking of features based on the amounts of leaks. The trace alignment in Profit is analogous to clustering in our technique to align traces of different secrets with similar timing profiles. Similarly, the packet-level features are analogous to extracting program internal features. The most important difference is the use-case: our model of systems is white-box and useful for defenders who have access to the systems. We consider the variations in both secret and public inputs, while the variations in Profit [56] is mostly related to secrets. While Profit could quantify information leaks, it can't find out what properties of secrets are leaking. We utilize program internal features and classifiers to localize code regions correlated with different observations and establish facts about leaking properties.

Side-channel Models. Chosen-message threats [2] where attackers can control public inputs are recently extended for different attack models [34], [59], [60]. Phan et al. [34] consider synthesizing adaptive side channels where in each step of the attack, the attacker chooses the best public input that maximizes the amount of information leaks. In our known-message threat model [2], however, the attacker only knows public inputs and may not control them to choose ideal public inputs. Many related works [34], [60], [7], [21] assume that the observations such as execution times are precise and apply abstractions such as the number of executed instructions. However, we support both realistic settings where the observations are noisy timing measurements and abstractions.

Quantification of information leaks. The amount of leaks can be estimated based on quantitative information flow [61], [9], [62], [63], [64]. Smith [9] defines min-entropy measure to quantify information leaks. With the assumption that the secret inputs are uniformly distributed and the program is deterministic, Smith [9] shows that the amount of information leaked based on the min-entropy is $log_2|L|$ where L is the classes of observations over the secret set. Our clustering algorithms can exploit the min-entropy measure defined by Smith [9] and give lower-bounds on the information leaks.

Localization of vulnerable code fragments. Machine learning techniques have been used to detect and pinpoint culprit codes [14], [16], [65]. Tizpaz-Niari et al. [16] consider performance issues in Java applications. They cluster the execution time of applications and then explain what program properties distinguish different functional clusters. The work [16] is limited to linear functions (as it needs to discover functions), while ours supports arbitrary timing functions over public inputs. In our security context, the program internal features can be functional. We use an extension of the decision tree algorithm in [16] to interpret different clusters. Symbolic executions have also been used to find vulnerable fragments [50], [66], [67]. Richer explanatory models are a unique aspect of our work. Our decision trees pinpoint basic blocks, contributing to different observations, as functions of public inputs.

IX. Threat to Validity

Overheads in Dynamic Analysis. We proposed a dynamic analysis approach to analyze functional side channels. Dynamic analysis often scales well to large applications. However, as compared with static analysis, they present additional overheads such as time required to discover variegated inputs and time needed for data collection.

Functional Regression and Order on Input Data. Our approach assumes the existence of an order over the public inputs to model timing functions. While such an order is natural for numerical variables, it may require ingenuity to define a suitable order for data types such as `strings` and `BitStream`. While our approach can work with any arbitrary user-defined ordering, often a suitable ordering can significantly improve the simplicity of the timing functions in the functional regression process. For instance, compare Figures 2 (d) and Figure 10 (d). Both of these applications model the leaks of set bits with different orders on the public inputs. Our approach captures the clusters in both examples, despite the ordering in Figure 10 (d) results in simpler functions. In practice, we restrict the functions explored in our regression to the class of basis-splines (B-splines). These models are parameterized by a given degree to model timing functions, and regression is more efficient with low-degree splines. In the case of higher-order target functions, we propose Gaussian Processes as an alternative to model timing functions.

Use of Decision Trees. The proposed decision tree models for discriminant learning partition the space of auxiliary features into hyper-rectangular sub-spaces. More expressive models, such as graph models, can be employed to learn richer classes of discriminants. However, we posit that simpler models like decision trees provide better interpretability. Another simplifying assumption in our approach is to model auxiliary features as functional attributes and map them to categorical labels. A more general approach would be to map the functions to numerical values and allow decision tree algorithms explore the space of features to identify suitable partitions. Further analysis of such mapping is left for future work.

Input Generations. Our approach requires a diverse set of inputs either given by users or generated automatically using the fuzzer. For instance, we used FUCHSIA to generate inputs for the *Regex* case study, while we use the inputs relevant to known vulnerabilities from DARPA STAC program for *SanpBuddy*. The quality of the debugging significantly depends on the presence of functional side channels in the given input set. Our fuzzing approach relies on heuristics to generate a diverse set of inputs, similar to existing evolutionary fuzzers.

Comparison with DifFuzz. We compared our approach against DifFuzz [7] in Section VI-C. We chose DifFuzz as an example of dynamic analysis tool with the point-wise definition of noninterference. We showed that the functional definition of noninterference gives a realistic sense of security. Since the clustering as the main tool for finding classes of observations took place after the input generations, the comparison may not evaluate the fuzzing engines accurately. We left combining fuzzing and clustering to detect the number of clusters during the input generations for future work.

Timing Measurements. The time observations in our case studies are measured on the NUC machine (see Section VI-B) to allow for higher precision in time and network observations. To further mitigate the effects of environmental factors such as Garbage Collections on timing measurements, we take the average of such measurements over multiple samples. In addition, we turned off JIT compiler for a better precision.

X. Conclusion and Future Work

We focused on the known-message setting under the assumption that secret inputs are less volatile than public inputs. In this setting, the observations appear as timing functions. We propose a notion of noninterference in the functional setting and show that it allows defenders to detect side channels using functional data clustering. We propose decision tree algorithms to pinpoint locations in the program that contribute to the side channels. Our tool FUCHSIA scales well for large real-world applications and aids debuggers to identify vulnerable fragments in such applications.

This work opens potential promising directions for future work. One direction is to combine the fuzzer with clustering that can directly estimate the number of distinguishable observations during the input generations. In this case, the objective is to find n secret values and m public values and maximize the number of distinguishable clusters in timing observations. Another direction is to study the potential timing side channels for machine learning applications. Given a learning problem with n samples and m features as public inputs, the feasibility of leaking (hyper-)parameter [68] of machine learning models via timing side channels is a relevant and challenging problem.

Acknowledgements. The authors would like to thank the anonymous reviewers for their valuable comments to improve our paper. This research was supported by DARPA under agreement FA8750-15-2-0096.

References

[1] S. Chen, R. Wang, X. Wang, and K. Zhang, "Side-channel leaks in web applications: A reality today, a challenge tomorrow," in *S&P*, 2010, pp. 191–206.

[2] B. Köpf and M. Dürmuth, "A provably secure and efficient countermeasure against timing attacks," in *CSF*. IEEE, 2009, pp. 324–335.

[3] "American fuzzy lop," 2016. [Online]. Available: http://lcamtuf.coredump.cx/afl/

[4] R. Kersten, K. Luckow, and C. S. Păsăreanu, "Poster: Afl-based fuzzing for java with kelinci," in *CCS*. ACM, 2017, pp. 2511–2513.

[5] J. Jacques and C. Preda, "Functional data clustering: a survey," *Advances in Data Analysis and Classification*, vol. 8, no. 3, pp. 231–255, 2014.

[6] F. Ferraty and P. Vieu, *Nonparametric functional data analysis: theory and practice*. Springer Science & Business Media, 2006.

[7] S. Nilizadeh, Y. Noller, and C. S. Păsăreanu, "Diffuzz: differential fuzzing for side-channel analysis," in *ICSE*, 2019, pp. 176–187.

[8] M. O. de la Fuente and M. Febrero-Bande, "Utilities for statistical computing in functional data analysis: The package fda. usc," 2011.

[9] G. Smith, "On the foundations of quantitative information flow," in *International Conference on Foundations of Software Science and Computational Structures*. Springer, 2009, pp. 288–302.

[10] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," in *Annual International Cryptology Conference*. Springer, 1996, pp. 104–113.

[11] "Timing side-channel on the password in eclipse jetty," May 2017, https://github.com/eclipse/jetty.project/blob/f3751d70787fd8ab93932a51c60514c2eb37cb58/jetty-util/src/main/java/org/eclipse/jetty/util/security/Credential.java#L81.

[12] "Timing attack in google keyczar library," 2009, https://rdist.root.org/2009/05/28/timing-attack-in-google-keyczar-library/.

[13] J. Ramsay, G. Hooker, and S. Graves, *Functional data analysis with R and MATLAB*. Springer Science & Business Media, 2009.

[14] S. Tizpaz-Niari, P. Černỳ, B.-Y. E. Chang, S. Sankaranarayanan, and A. Trivedi, "Discriminating traces with time," in *TACAS*. Springer, 2017, pp. 21–37.

[15] S. Chiba, "Load-time structural reflection in java," in *European Conference on Object-Oriented Programming*. Springer, 2000, pp. 313–336.

[16] S. Tizpaz-Niari, P. Černý, B. E. Chang, and A. Trivedi, "Differential performance debugging with discriminant regression trees," in *32nd AAAI Conference on Artificial Intelligence*, 2018, pp. 2468–2475.

[17] T. Górecki and M. Łuczak, "First and second derivatives in time series classification using dtw," *Communications in Statistics-Simulation and Computation*, vol. 43, no. 9, pp. 2081–2092, 2014.

[18] J. A. Goguen and J. Meseguer, "Security policies and security models," in *IEEE S&P*, 1982, pp. 11–11.

[19] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on selected areas in communications*, vol. 21, no. 1, pp. 5–19, 2003.

[20] T. Terauchi and A. Aiken, "Secure information flow as a safety problem," in *International Static Analysis Symposium*. Springer, 2005, pp. 352–367.

[21] J. Chen, Y. Feng, and I. Dillig, "Precise detection of side-channel vulnerabilities using quantitative cartesian hoare logic," in *CCS*, 2017, pp. 875–890.

[22] J. O. Ramsay, *Functional data analysis*. Wiley Online Library, 2006.

[23] R. Alur and N. Singhania, "Precise piecewise affine models from input-output data," ser. EMSOFT, 2014, pp. 3:1–3:10.

[24] S. C. Johnson, "Hierarchical clustering schemes," *Psychometrika*, vol. 32, no. 3, pp. 241–254, 1967.

[25] K. Wagstaff, C. Cardie, S. Rogers, S. Schrödl *et al.*, "Constrained k-means clustering with background knowledge," in *ICML*, 2001, pp. 577–584.

[26] J. Song, H. Wang, and M. J. Song, "Package ckmeans," 2017.

[27] I. Davidson and S. Ravi, "Clustering with constraints: Feasibility issues and the k-means algorithm," in *2005 SIAM international conference on data mining*. SIAM, 2005, pp. 138–149.

[28] R Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2013, ISBN 3-900051-07-0. [Online]. Available: http://www.R-project.org/

[29] L. Breiman, J. Friedman, R. Olshen, and C. Stone, *Classification and regression trees*. Wadsworth: Belmont, CA, 1984.

[30] "Intel nuc5i5ryh," https://ark.intel.com/content/www/us/en/ark/products/83255/intel-nuc-kit-nuc5i5ryh.html.

[31] M. Febrero-Bande and M. Oviedo de la Fuente, "Statistical computing in functional data analysis: The R package fda.usc," *Journal of Statistical Software*, vol. 51, no. 4, pp. 1–28, 2012. [Online]. Available: http://www.jstatsoft.org/v51/i04/

[32] S. Chiba, "Javassist - a reflection-based programming wizard for java," in *Proceedings of OOPSLA Workshop on Reflective Programming in C++ and Java*, vol. 174, 1998.

[33] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[34] Q.-S. Phan, L. Bang, C. S. Pasareanu, P. Malacaria, and T. Bultan, "Synthesis of adaptive side-channel attacks," in *CSF*. IEEE, 2017, pp. 328–342.

[35] Apogee-Research, "Guess secret version 2," 2017, https://github.com/Apogee-Research/STAC/blob/masterCanonical_Examples/Source/Category3_vulnerable.java.

[36] "Timing side-channel on the length of password in eclipse jetty," May 2017, https://github.com/eclipse/jetty.project/commit/2baa1abe4b1c380a30deacca1ed367466a1a62ea.

[37] "Fixed timing side-channel on the length of password in eclipse jetty," August 2017, https://github.com/eclipse/jetty.project/commit/a7e8b4220a410b85c843bffcd13f07d70f1b3fe8.

[38] "Updated secret key comparison in openjdk-8-u," https://hg.openjdk.java.net/jdk8u/jdk8u-dev/jdk/file/1832c29655eb/src/share/classes/javax/crypto/spec/SecretKeySpec.java.

[39] "The comparison of byte arrays in openjdk-8-u," https://hg.openjdk.java.net/jdk8u/jdk8u-dev/jdk/file/1832c29655eb/src/share/classes/java/security/MessageDigest.java.

[40] Apogee-Research, "Stegosaurus application," 2017, https://github.com/Apogee-Research/STAC/.

[41] "Snapbuddy application," 2016, https://github.com/Apogee-Research/STAC/tree/master/Engagement_Challenges/Engagement_2/snapbuddy_1.

[42] J. Agat, "Transforming out timing leaks," in *POPL*. ACM, 2000, pp. 40–53.

[43] H. Mantel and A. Starostin, "Transforming out timing leaks, more or less," in *ESORICS*. Springer, 2015, pp. 447–467.

[44] J. B. Kruskal, "On the shortest spanning subtree of a graph and the traveling salesman problem," *Proceedings of the American Mathematical society*, vol. 7, no. 1, pp. 48–50, 1956.

[45] "Kruskal's algorithm implementations," https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/.

[46] Apogee-Research, "Collab application," 2017, https://github.com/Apogee-Research/STAC/tree/master/Engagement_Challenges/Engagement_4/collab.

[47] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying constant-time implementations." in *USENIX Security Symposium*, 2016, pp. 53–70.

[48] T. Antonopoulos, P. Gazzillo, M. Hicks, E. Koskinen, T. Terauchi, and S. Wei, "Decomposition instead of self-composition for proving the absence of timing channels," in *PLDI*, vol. 52, no. 6. ACM, 2017, pp. 362–375.

[49] G. Doychev, B. Köpf, L. Mauborgne, and J. Reineke, "Cacheaudit: A tool for the static analysis of cache side channels," *ACM Transactions on Information and System Security (TISSEC)*, vol. 18, no. 1, p. 4, 2015.

[50] S. Wang, P. Wang, X. Liu, D. Zhang, and D. Wu, "Cached: Identifying cache-based timing channels in production software," in *26th USENIX Security Symposium*, 2017, pp. 235–252.

[51] P. Gao, J. Zhang, F. Song, and C. Wang, "Verifying and quantifying side-channel resistance of masked software implementations," *TOSEM*, vol. 28, no. 3, 2019.

[52] G. Barthe, P. R. D'Argenio, and T. Rezk, "Secure information flow by self-composition," in *CSF*. IEEE, 2004, pp. 100–114.

[53] V. B. Livshits and M. S. Lam, "Finding security vulnerabilities in java applications with static analysis." in *USENIX Security Symposium*, vol. 14, 2005, pp. 18–18.

[54] D. Landman, A. Serebrenik, and J. J. Vinju, "Challenges for static analysis of java reflection-literature review and empirical study," in *ICSE*. IEEE, 2017, pp. 507–518.

[55] D. Milushev, W. Beck, and D. Clarke, "Noninterference via symbolic execution," in *Formal Techniques for Distributed Systems*. Springer, 2012, pp. 152–168.

[56] N. Rosner, I. Burak Kadron, L. Bang, and T. Bultan, "Profit: Detecting and quantifying side channels in networked applications," *NDSS*, 2019, https://www.ndss-symposium.org/wp-content/uploads/2019/02/ndss2019_05B-2_Rosner_paper.pdf.

[57] S. Tizpaz-Niari, P. Černý, and A. Trivedi, "Quantitative mitigation of timing side channels," in *Computer Aided Verification (CAV)*, 2019, pp. 140–160.

[58] S. Tizpaz-Niari, P. Černý, S. Sankaranarayanan, and A. Trivedi, "Efficient detection and quantification of timing leaks with neural networks," in *Runtime Verification (RV)*, 2019, pp. 329–348.

[59] L. Bang, A. Aydin, Q.-S. Phan, C. S. Păsăreanu, and T. Bultan, "String analysis for side channels with segmented oracles," in *FSE'16*. ACM, 2016, pp. 193–204.

[60] C. S. Pasareanu, Q.-S. Phan, and P. Malacaria, "Multi-run side-channel analysis using symbolic execution and max-smt," in *CSF*. IEEE, 2016, pp. 387–400.

[61] B. Köpf and D. Basin, "An information-theoretic model for adaptive side-channel attacks," in *CCS*. ACM, 2007, pp. 286–296.

[62] M. Backes, B. Köpf, and A. Rybalchenko, "Automatic discovery and quantification of information leaks," in *IEEE S&P*. IEEE, 2009, pp. 141–153.

[63] B. Köpf and G. Smith, "Vulnerability bounds and leakage resilience of blinded cryptography under timing attacks," in *CSF*. IEEE, 2010, pp. 44–56.

[64] T. Chothia, Y. Kawamoto, and C. Novakovic, "A tool for estimating information leakage," in *CAV*. Springer, 2013, pp. 690–695.

[65] L. Song and S. Lu, "Statistical debugging for real-world performance problems," in *OOPSALA, 2014*, vol. 49, no. 10. ACM, 2014, pp. 561–578.

[66] S. Guo, M. Wu, and C. Wang, "Adversarial symbolic execution for detecting concurrency-related cache timing leaks," in *ESEC/FSE*. ACM, 2018, pp. 377–388.

[67] M. Wu, S. Guo, P. Schaumont, and C. Wang, "Eliminating timing side-channel leaks using program repair," in *ISSTA*. ACM, 2018, pp. 15–26.

[68] B. Wang and N. Z. Gong, "Stealing hyperparameters in machine learning," in *IEEE Symposium on Security and Privacy*. IEEE, 2018, pp. 36–52.