QFuzz: Quantitative Fuzzing for Side Channels

Yannic Noller
yannic.noller@acm.org
National University of Singapore
Singapore

Saeid Tizpaz-Niari
saeid@utep.edu
University of Texas at El Paso
USA

ABSTRACT

Side channels pose a significant threat to the confidentiality of software systems. Such vulnerabilities are challenging to detect and evaluate because they arise from non-functional properties of software such as execution times and require reasoning on multiple execution traces. Recently, *noninterference* notions have been adapted in static analysis, symbolic execution, and greybox fuzzing techniques. However, noninterference is a strict notion and may reject security even if the strength of information leaks are weak. A quantitative notion of security allows for the relaxation of noninterference and tolerates small (unavoidable) leaks. Despite progress in recent years, the existing quantitative approaches have scalability limitations in practice.

In this work, we present QFuzz, a greybox fuzzing technique to quantitatively evaluate the strength of side channels with a focus on *min entropy*. Min entropy is a measure based on the number of distinguishable observations (partitions) to assess the resulting threat from an attacker who tries to compromise secrets in one try. We develop a novel greybox fuzzing equipped with two partitioning algorithms that try to maximize the number of distinguishable observations and the cost differences between them.

We evaluate QFuzz on a large set of benchmarks from existing work and real-world libraries (with a total of 70 subjects). QFuzz compares favorably to three state-of-the-art detection techniques. QFuzz provides quantitative information about leaks beyond the capabilities of all three techniques. Crucially, we compare QFuzz to a state-of-the-art quantification tool and find that QFuzz significantly outperforms the tool in scalability while maintaining similar precision. Overall, we find that our approach scales well for real-world applications and provides useful information to evaluate resulting threats. Additionally, QFuzz identifies a zero-day side-channel vulnerability in a security critical Java library that has since been confirmed and fixed by the developers.

CCS CONCEPTS

• **Security and privacy → Software security engineering**; • **Software and its engineering → Software testing and debugging**.

KEYWORDS

vulnerability detection, side-channel analysis, quantification, dynamic analysis, fuzzing

ISSTA '21, July 11–17, 2021, Virtual, Denmark
© 2021 Association for Computing Machinery.
This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '21), July 11–17, 2021, Virtual, Denmark*, https://doi.org/10.1145/3460319.3464817.

ACM Reference Format:
Yannic Noller and Saeid Tizpaz-Niari. 2021. QFuzz: Quantitative Fuzzing for Side Channels. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '21), July 11–17, 2021, Virtual, Denmark*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3460319.3464817

1 INTRODUCTION

Side-channel (SC) vulnerabilities allow attackers to compromise secret information by observing runtime behaviors such as response time, cache hit/miss, memory consumption, network packet, and power usage. Software developers are careful to prevent malicious eavesdroppers from accessing secrets using techniques such as encryption. However, these techniques often fail to guarantee security in the presence of side channels since they arise from non-functional behaviors and require simultaneous reasoning over multiple runs.

Side-channel attacks remain a challenging problem even in security-critical applications. There are known practical side-channel attacks against the RSA algorithm [7], an online health system [10], the Google's Keyczar Library [24], and the Xbox 360 [37]. In the Xbox 360, timing side channels allowed attackers to reduce the maximum number of trials to compromise a 16 byte secret from 256^{16} to $256 * 16$ due to vulnerable implementations in byte array comparisons. Another example is Spectre [20] that challenged the confidentiality of computer devices via side channels.

Recently, techniques have been developed to detect side-channel vulnerabilities in software [1, 9, 27]. These works have shown notable success in finding critical vulnerabilities in libraries such as Eclipse Jetty, Apache Ftp, and OpenJDK Crypto. Despite these discoveries, the detection techniques often rely on the *noninterference* notion of security, which leads to binary answers with limited extra information such as the maximum timing difference observed to reject the security [27]. A quantitative notion is crucial in the security evaluation of real-world applications. The notion allows developers to evaluate the resulting threat of side channels precisely and even relax noninterference to tolerate small leaks, which might be necessary in practice. Previous work has also used quantitative methods to evaluate the strength of information leaks [3, 29, 32], however, they are limited to small programs and do not scale well for large applications and input spaces.

Prevalent quantification measures such as Shannon entropy [3, 21] show the expected amount of leaks over an unbounded number of trials and fail to evaluate the resulting threats for a more practical setting where an attacker can try one ideal guess. In this paper, we focus on the immediate threats that allow an attacker to guess the secret correctly in one try. In this threat model, Smith [32] showed that the number of distinguishable observations (i.e., partitions) precisely quantifies the strength of information leaks. The corresponding quantitative notion is known as *min entropy*.

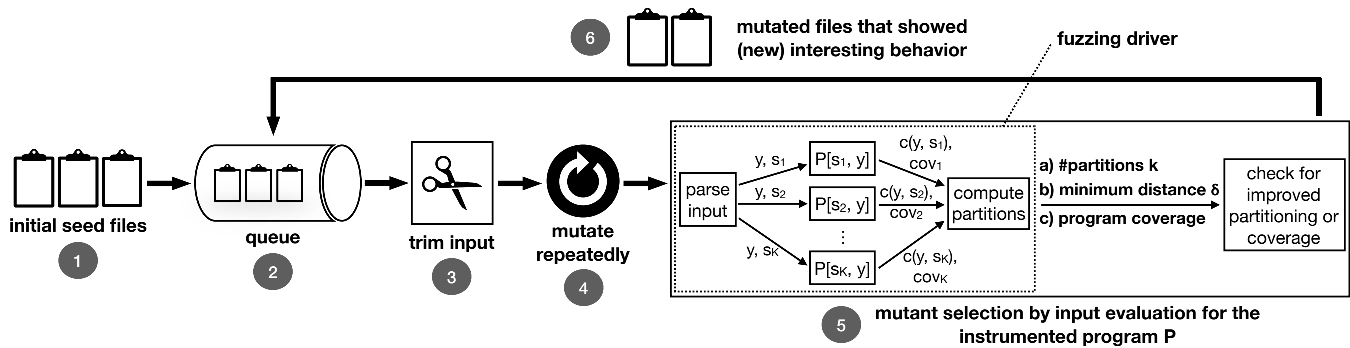

Figure 1: Workflow of QFuzz.

We introduce a practical variant of min entropy that can tolerate ϵ cost differences in introducing a new partition. Then, we adapt a greybox evolutionary algorithm to approximate the number of partitions with lower-bound guarantees. To the best of our knowledge, this is the first work to adapt greybox evolutionary fuzzing to characterize quantitative measures of information leaks that scale well for large applications, handle dynamic features, and provide lower-bound guarantees. In particular, we extend DifFuzz [27], a greybox fuzzer for side-channel detection, with partitioning algorithms to quantify the amount of leaks. In addition to code coverage measures, our algorithm guides the search to find as many secret values along with single public value such that the following criteria are maximized: (1) the number of partitions and (2) the cost differences between partitions.

We propose two partitioning algorithms to find distinguishable classes and the cost distance between them. The first algorithm takes a dynamic programming approach and maximizes both criteria, but it may not provide bounds on the estimation of information leaks. The second partitioning algorithm is greedy, which has cheaper computations and provides a lower-bound guarantee, but it may not maximize the cost differences between partitions.

We implement our approach in a tool named QFuzz and apply QFuzz on 70 subjects including real-world Java libraries. On the set of benchmarks, we first compare the two partitioning algorithms. Then, we compare our approach against state-of-the-art detection tools. The result shows that QFuzz has better scalability and detection when compared to Themis [9] and Blazer [1], while it is similar to DifFuzz [27]. QFuzz also provides quantitative information that is useful to evaluate threats, beyond the capabilities of Themis, Blazer, and DifFuzz. Finally, we compare our approach to the quantification technique MaxLeak [29]. QFuzz outperforms MaxLeak in scalability while providing comparable precision.

On a set of real-world benchmarks, we show the scalability and usefulness of our approach. QFuzz discovers a zero-day side-channel vulnerability in Apache WSS4J, a library for the implementation of Web Services Security. This vulnerability has been confirmed and fixed by developers [35].

Our main contributions are:

- a novel greybox fuzzing approach to *detect* and *quantify* side-channel vulnerabilities by characterizing the min entropy,
- the publicly available implementation of QFuzz, and
- the evaluation of QFuzz by showing its scalability and usefulness when compared to state-of-the-art techniques as well as in real-world applications, including the discovery of a previously unknown vulnerability in a security-critical library.

2 OVERVIEW

QFuzz in a Nutshell. QFuzz searches for a single public input and K secret inputs that maximize the number of partitions, as well as the distance between these partitions. Intuitively, a pair of secret inputs is distinguishable and belongs to different partitions if their cost differences are greater than a tolerance parameter ϵ. As illustrated in Figure 1, QFuzz uses an evolutionary greybox fuzzing approach to evolve the public and secret values. It starts with some initial seed files and applies random mutations like random bit flips and crossovers. To assess the quality of the evolved inputs, the fuzzing driver parses the inputs into a public value y and K secret values $(s_1, s_2, ..., s_K)$, and executes the instrumented program with the corresponding input pairs. The program's instrumentation keeps track of the resource usage (e.g., number of executed instructions, or memory consumption), and provides the actual observation, denoted as the function c in Figure 1. Afterward, the observations are processed and the corresponding partitions are computed. Finally, the fitness function decides based on the number of partitions in the side-channel observations, the minimum distance δ between these partitions, and the overall coverage whether it is interesting to keep the input.

Example. We use the password matching implementations from Eclipse Jetty as well as Spring-Security to illustrate our approach. Figure 2 shows three different variants of Jetty and an unsafe variant of Spring-Security. The first variant was vulnerable to timing side channels since it used Java internal string equality to compare a (secret) password against a given (public) guess as shown in Figure 2 (top-left). The developers made multiple fixes and finally committed the final fix [12] (the current implementation) as shown in Figure 2 (top-right). The implementation in Figure 2 (bottom-left) is another candidate for password matching, taken from OpenJDK library [28].

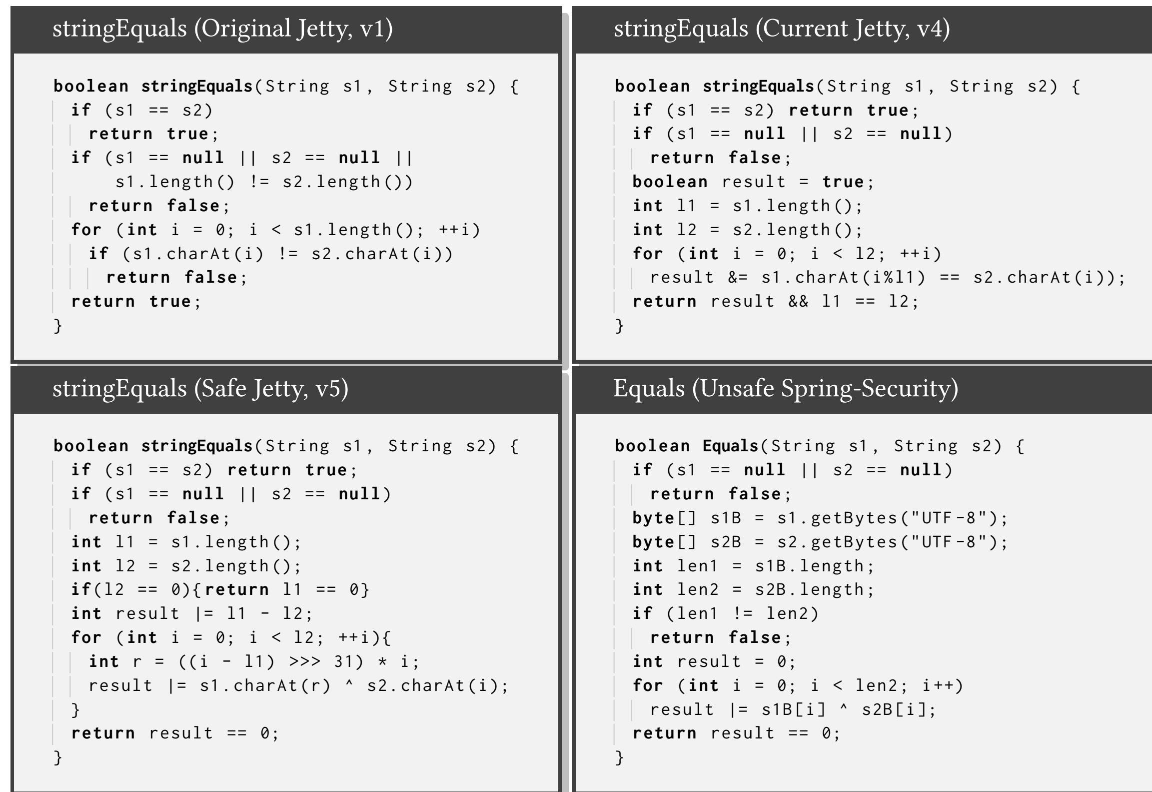

Figure 2: String equality in Eclipse Jetty (`s1` secret, `s2` public). Top-Left: The code snippet is the original implementation for the secret comparison that contains a strong side channel. Top-Right: The code is the current version that has been developed to fix the side channel, but still leaks some information. Bottom-Left: The code snippet is a proposed safe implementation. Bottom-Right: String equality in Spring-Security that leak whether the length of strings is matching.

Finally, we consider an unsafe variant of password matching from Spring-Security as shown in Figure 2 (bottom-right). We study the feasibility of side channels in these four implementations and apply QFuzz to estimate the amount of information leaks using the number of partitions.

Example Parameters. We consider $K = 100$ and $\epsilon = 1$ as default configuration parameters. We set the length of the secret and the public guess to be the same and fixed to 16 characters. We run QFuzz 30 times on each variant, where each run is for 30 minutes. We report the maximum number of partitions (k) and the cost differences in bytecode between two closest partitions (δ). The detailed results for can be found in Table 1 and Table 3.

First Variant of Jetty. Figure 2 (top-left) shows the first variant, for which QFuzz discovers 17 classes of observations ($k = 17$). Each partition is at least 3 bytecodes far from any other partition ($\delta = 3$). Since we fix the length, the number of partitions reflect side-channel observations related to the content of secret inputs. We find that each partition shows the number of characters in the prefix of secrets that match with the guess. Since there are 16 characters, there can be 17 partitions ranging from no prefix match to all 16 characters match. This implementation is known to be vulnerable to adaptive side channels where an attacker can use the cost observations to compromise a prefix of a secret password in each step of the attack. The outcome of QFuzz indirecly indicates the feasibility of adaptive attacks, while they are not the main focus.

Second Variant of Jetty (current implementation). We consider the current implementation in Jetty as shown in Figure 2 (top-right). In this case, QFuzz detects 9 partitions where each partition is at least 1 bytecode far from any others. This analysis shows that the fix improved the security and reduced the strength of leaks as compared to the first variant. Since QFuzz found multiple partitions, however, we conclude that this variant is not completely safe. To understand the issue, we analyze the corresponding instructions generated by Java Virtual Machine (JVM). The analysis shows the equal operator ("==") in the loop body is optimized by JVM and translated to a conditional jump instruction (`if_icmpne`) if the comparison is not successful and an unconditional jump instruction otherwise. This translation introduces an imbalance comparison where the unconditional jump includes a single extra bytecode instruction as compared to the conditional jump. With 16 characters, the bytecode differences, range from 0 to 16, are partitioned into 9 classes with $\epsilon = 1$.

Third Variant of Jetty (OpenJDK [28]). We take a password matching algorithm from OpenJDK [28] that explicitly uses "xor" operation

instead of "==" as depicted in Figure 2 (bottom-left). We apply QFuzz to this implementation and obtain only 1 partition. Therefore, we deem this variant a completely safe implementation for the password matching. Looking into the instruction bytecodes, we did not find any conditional or unconditional jump in the loop body. The JVM directly uses the xor operation for the comparison.

Unsafe Variant of Spring-Security. We study an unsafe string matching algorithm from Themis [9], especially to compare our approach with the state-of-the-art. As shown in Figure 2 (bottom-right) the length of secret and public strings are compared after encoding with the `String.getBytes("UTF-8")` function. We apply QFuzz to this implementation, and it reported 2 partitions with $\delta = 149$. With a closer look into the identified partitions, the secret values in one partition contain a special character, which is then mapped to two byte values. This indicates that there is an early return of length mismatch over bytecode encodings even if the string lengths of secret and public inputs are the same.

Comparison to Related Work. The state-of-the-art *detection* techniques such as DifFuzz [27] rely on the noninterference that deems applications safe or unsafe with extra information such as the maximum cost difference between two secret inputs, which are not as useful as the number of partitions. We use the example of Spring-Security, Figure 2 (bottom-right), to show differences concretely. QFuzz identifies 2 partitions with $\delta = 149$. Similarly, DifFuzz classifies the subject as unsafe with the cost differences of 149 bytecodes. At first glance, the results look similar. But, there are subtle differences. DifFuzz implies a strong side channel by showing that the maximum differences between two secret values are 149 bytecodes. QFuzz indicates that there are only two partitions that are 149 bytcodes far from each other, and there is no other observation between them with cost differences more than one bytecode ($\epsilon = 1$). Thus, QFuzz implies that the strength of leak is weak and is unlikely to compromise whole secret. In fact, the side channel only leaks whether the secret string contains a special character or not.

Additionally, the state-of-the-art *quantitative* techniques for side channels like MaxLeak [29] are often developed based on static analysis or symbolic execution, which may not scale for practical examples as shown in Section 5.6.

Summary. Our analysis provides an under-approximation of the (true) number of partitions via side-channel observations. As shown in this example, QFuzz provides quantitative information that can be useful to evaluate security among multiple variants, understand the strength of leaks, and identify safer implementations that meet the security requirements.

3 PROBLEM DEFINITION

First, we define the resource usage model of programs:

Definition 3.1 (Cost Model). *The cost model of a deterministic program $\mathcal{P}$ is a tuple $[[\mathcal{P}]] = (X, Y, \Sigma, c)$ where $X = \{x_1, \ldots, x_n\}$ is the set of secret inputs, $Y = \{y_1, \ldots, y_m\}$ is the set of public inputs, $\Sigma \subseteq \mathbb{R}^n$ is a finite set of secrets, and $c : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}_{\geq 0}$ is the cost function of the program over the secret and public inputs.*

We assume that the abstraction of execution times as the number of executed Java bytecodes is precise and corresponds to the actual timing observations. Therefore, we overwrite the cost function to be $c : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{N}_{>0}$. Our goal is to characterize the information leaks in a given program with the quantitative measures. Thus, we consider the finite set of secret inputs.

Smith [32] noted that prevalent quantitative metrics such as Shannon entropy characterize the expected resulting threats over unbounded number of trials. Instead, Smith proposed *min* entropy that reflects the resulting threat from an attacker who can try one best guess. To see the differences, let us consider the timing models of two programs $T_A = H \mid 31$ and $T_B = H \,\&\, L$ where H and L are 6 bits (uniform) secret and public inputs, respectively. Program A leaks the most significant bit, and program B leaks zero to all bits depending on L. *Shannon* entropy [21] quantifies the leaks to be 1 bit and 3 bits for the first and second variants, respectively. On the other hand, min entropy [32] uses the maximum partitions and quantifies the leaks to be $\log_2 2 = 1$ bit and $\log_2 64 = 6$ bits for the first and second variants, respectively! We note that 64 partitions can be observed with the best guess of $L = 63$ in the program B.

Threat Model. We consider a chosen-message threat setting [22] where an attacker can pick an ideal public input to compromise the secret value or some properties of it in one try. In the offline mode, the attacker, who has access to the source code, samples execution times using the ideal public input and partitions secret values into different classes of observations. During the online phase of the attack, the attacker queries the target application with the public input and tries to compromise the fixed secret by mapping the observed time to a partition of secret values. We assume that the precision in the attacker's observation can be set with a tolerance parameter ϵ such that any cost differences more than ϵ is distinguishable.

Proposition 3.1 (Smith [32]). *Let $\Sigma_{Y=y} = \langle S_1, S_2, \ldots, S_k \rangle$ be the quotient space of Σ characterized by the cost observations under the public input y such that any pair of secret values are in the partition i ($s, s' \in S_i$ for any $1 \leq i \leq k$), if and only if $c(s, y) = c(s', y)$. Let $Y = y^*$ be a single public input that gives the maximum number of classes in the cost observations k^*, i.e., $|\Sigma_{Y=y^*}| \geq |\Sigma_{Y=y}|$ for any $y \in Y$. Assuming that the program $\mathcal{P}$ is deterministic and the distribution over secret input Σ is uniform, then amounts of information leaked according to min-entropy measure can be characterized with $log_2 k^*$.*

Proposition 3.1 characterizes min entropy with the exact cost equality, i.e., $\epsilon = 0$. However, the exact equality is a strict security policy, and it is prevalent to account for uncertainties in observations with $\epsilon > 0$ [9, 27, 34]. Since a positive value of ϵ does not characterize the equivalence class over Σ, we adapt the proposition 3.1 with a relation that is reflexive and symmetric, but not transitive. The relation is known as tolerance relation [17, 30].

Proposition 3.2 (Maximal set of similar secrets [17]). *Given any tolerance $\epsilon \geq 0$, a quotient space of $\Sigma_{Y=y} = \langle S_1, S_2, \ldots, S_k \rangle$ characterizes the maximal tolerance classes if and only if (1) $s, s' \in S_i$ (for $1 \leq i \leq k$), if and only if $|c(s, y) - c(s', y)| \leq \epsilon$ and (2) $\forall s \notin S_i$ (for $1 \leq i \leq k$), $\exists s' \in S_i$, $|c(s, y) - c(s', y)| > \epsilon$.*

It is straightforward to prove that the number of maximal tolerance classes is equal to the number of distinguishable observations for an attacker with the tolerance parameter $\epsilon \geq 0$. In addition to

Algorithm 1: QFuzz: Evolutionary fuzzing for quantifying information leakage.

Input: Program $\mathcal{P}$, initial seed IS, partitioning algorithm $Part$, tolerance parameter ϵ, timeout T, max. num. partitions K.
Output: num. partitions k, distance δ

1. $k, \delta, population \leftarrow 1, 0, IS$
2. $inputs \leftarrow$ `mutation_pick`$(population)$
3. $y, s_1, \ldots, s_K \leftarrow$ `parse`$(inputs, constraints)$
4. $(cost_1, p_1), \ldots, (cost_K, p_K) \leftarrow \mathcal{P}(s_1, y), \ldots, \mathcal{P}(s_K, y)$
5. $k', \delta' \leftarrow \texttt{Part}_\epsilon(cost_1, \ldots, cost_K)$
6. **if** $k' > k$ ***or*** $(k = k'$ ***and*** $\delta' > \delta)$ **then**
7.   Add $(y, s_1, \ldots, s_K)$ to population
8. $k, \delta \leftarrow k', \delta'$
9. **else if** *any path* p_i *characterizes a new path* **then**
10.   Add (y, s_i) to population
11. **if** $t \leq T$ **then**
12.   Go to 2
13. **else**
14.   **return** k, δ.

the number of observations, the cost differences between distinguishable observations are critical to compromise secrets in one trial. Given two public inputs y_1, y_2 that characterize the same number of distinguishable observations k, we measure the distance between observations with

$$\xi(\Sigma, y) = \sum_{i=1}^{k} \min_{j=i+1}^{k} \left\{ |c(s, y) - c(s', y)| : s \in S_i \wedge s' \in S_j \right\}$$

and pick y_1 over y_2 if and only if $\xi(\Sigma_1, y_1) > \xi(\Sigma_2, y_2)$.

Definition 3.2 (Problem Statement). *Given a deterministic program $\mathcal{P}$ with secret inputs $X \in \mathbb{R}^n$ and public inputs $Y \in \mathbb{R}^m$, a cost function $c \in \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{N}_{>0}$, and a tolerance $\epsilon \in \mathbb{N}$, the key computational problem is to find a finite set of secret values Σ and a public input y^* that characterizes the maximum number of classes in the observation with the highest distances, i.e., $|\Sigma_{Y=y^*}| > |\Sigma_{Y=y}|$ or $|\Sigma_{Y=y^*}| = |\Sigma_{Y=y}| \wedge \xi(\Sigma, y^*) \geq \xi(\Sigma, y)$ for any $y \in Y$, and compute min entropy as $log_2|\Sigma_{Y=y^*}|$.*

The problem of finding inputs to characterize the maximum number of distinguishable observations with the highest distance is hard. First, the input generation part requires an exhaustive search in the exponential set of subsets of input space, and hence is clearly intractable. Given a set of secret inputs Σ, and public input y, finding the number of distinguishable observations with a high distance is NP-complete. The proof can establish from Graph K-Colorability problem that is known to be NP-Complete for any $K \geq 3$.

4 QUANTIFYING LEAKS WITH FUZZING

Since the problem 3.2 is hard, we propose a dynamic approach to approximate the min entropy with lower-bound guarantees. In particular, we develop an evolutionary fuzzing algorithm with constrained partitioning.

General Algorithm. Algorithm 1 shows the high-level steps in our quantitative fuzzing approach. A high-level illustration is shown in Figure 1. It starts with picking an input from the population and performs some mutations. For this step, we build on top of existing fuzzing approaches like AFL [38], which includes, e.g., random bit flips, random deletions/insertions, and crossovers. After parsing the inputs, QFuzz runs the program on the set of secret and public inputs and characterizes the paths and costs. Then, it applies the partitioning algorithm with the given tolerance parameter and returns the number of partitions and the distance between them. The fuzzer will keep an input if the values characterize more distinguishable observations or a higher distance between the partitions. Similar to prevalent evolutionary fuzzing, inputs with new path coverage are still added to the population.

Objective Function. The fuzzing objective is to find a public input and a set of secret inputs such that the number of distinguishable observations over the set of secret values is maximized. Let $\epsilon \in \mathbb{N}_{\geq 0}$ be the tolerance parameter to distinguish secret values and K be an upper-bound on the number of distinguishable observations. The quantitative fuzzing aims to maximize the following objective and returns corresponding set of secrets and a public input:

$$\max_{s_1,\ldots,s_K,y} \left|\texttt{Part}_\epsilon\big(c(s_1, y), \ldots, c(s_K, y)\big)\right| + \big(1 - \exp(-0.1 * \delta)\big)$$

where the partitioning function $\texttt{Part}_\epsilon$ returns the distinguishable classes of observations (partitions) from the generalized set of K secret values and a public input y under the tolerance parameter ϵ. The distance parameter δ is measured by the $\texttt{Part}_\epsilon$ function. The term $1 - \exp(-0.1 * \delta)$ is always between 0 and 1, and prefers a partitioning with the maximum distance δ over any other partitioning with the same number of classes. Thus, the objective function of our fuzzer guides the search for finding interesting inputs with two criteria: (1) the number of discovered partitions is maximized and (2) the cost differences (distance) between partitions are maximized.

We combine our custom objective function with the code coverage criteria during fuzzing. In this way, our approach can circumvent local minima in the search space: if we cannot identify more partitions, our search will still attempt to increase program coverage and continue to explore other parts of the program.

The partitioning algorithm $\texttt{Part}_\epsilon$ is a key component in our fuzzing approach. We propose two algorithms for efficient partitioning, which both have their merits: *KDynamic* and *Greedy*. The *KDynamic* algorithm (see Section 4.1) adapts a dynamic programming approach to find classes that have large distances. This algorithm, however, is expensive and may over-approximate the number of distinguishable classes. The *Greedy* algorithm (see Section 4.2) performs a greedy selection of partitions. This algorithm is fast and finds the exact number of partitions, but it may not find a partitioning with the maximum distance between classes.

4.1 Partitioning with Dynamic Programming

Our *KDynamic* algorithm is inspired by ideas from computing optimal histograms [15]. Let $T = \{c_1, \ldots, c_K\}$ be the set of positive integers (costs) that is indexed in ascending order, i.e., $c_i < c_j$ whenever $i < j$. Let $T_r = \{c_1, \ldots, c_r\} \subseteq C$ be the set of costs up to the index $r \leq K$. An $k-$partition of T is $B_k = \{b_1, \ldots, b_k\}$ where each class i is $\{c \in T | b_{i-1} < c \leq b_i\}$. We define the cost of partitioning to be the sum of differences between the smallest and largest costs in each class. Formally, $cost(B) = \sum_{i=1}^{k} |\max\{c | c \in b_i\} - \min\{c | c \in b_i\}|$.

It follows from [15, 22] that every optimal partitioning of T_r contains an optimal partitioning for some T_q with $q < r$. With this optimal substructure property, we can now define the dynamic programming to construct optimal partitioning. Let $a(r, i)$ be the cost of partitioning for T_r and i classes defined as the following:

$$a(r, 1) = |c_r - c_1|$$

$$a(r, i) = \min_{1 \leq j < r} \left\{ a(j, i-1) + |c_r - c_{j+1}| \right\}$$

For a given k, the worst-case running time for this algorithm is $O(k.K^2)$. We invoke this algorithm for each $k = 1, \ldots, K$ in this order and find the smallest k to satisfy the constraint $\forall b_i \in B_k, \forall s, s' \in b_i, |c(s, y) - c(s', y)| \leq \epsilon$, with the distance between partitions is: $\sum_{i=1}^{k-1} |B_i.max - B_{i+1}.min|$ for $k>1$.

4.2 Partitioning with Greedy Approach

Our *Greedy* partitioning is shown in Algorithm 2. After preprocessing the set of cost observations, *Greedy* picks the observation with the lowest cost and puts all other observations (in the sorted order) that are ϵ-close to the lowest observation in the first class. Next, *Greedy* picks the lowest cost observation that is not covered in the first class and puts all ϵ-close observations in the second class. This procedure continues until all costs are covered and assigned to a class. The complexity of algorithm is equal to the complexity of the sorting algorithm that is $O(K. \log(K))$.

Algorithm 2: Partitioning with a greedy approach.

```
Input: Cost T = {c_1, ..., c_K}, threshold ε
Output: num. partitions k, distance δ
1  remove_duplicates_sort(T)
2  B, b, δ ← {}, ∅, 0
3  for each c ∈ T do
4      if b = ∅ or (|c − b.min| ≤ ε) then
5          b.add(c)
6      else
7          B.add(b)
8          b ← {c}
9  B.add(b)
10 if |B| ≥ 2 then
11     δ ← Σ_{i=1}^{|B|−1} |B_i.max − B_{i+1}.min|
12 return |B|, δ
```

5 EVALUATION

We evaluate QFuzz on a large set of benchmarks and focus thereby on the following three research questions:

- **RQ1** Which partitioning algorithm (*KDynamic* or *Greedy*) performs better in terms of correct number of partitions and time for partition computation?
- **RQ2** How does QFuzz compare with state-of-the-art SC detection techniques like Blazer [1], Themis [9], and DifFuzz [27]?
- **RQ3** Can QFuzz be used for the quantification of SC vulnerabilities in real-world Java applications and how does it compare with MaxLeak [29]?

All subjects, experimental results, and our tool are available on our GitHub repository: https://github.com/yannicnoller/qfuzz

5.1 Subjects

For RQ1, we use a micro-benchmark and real-world Java programs. We compare the partitioning algorithms on the variants of *Eclipse Jetty* presented in Section 2 and a set of *Leak Set* programs that are leaking the number of set bits. In *Eclipse Jetty*, the secret inputs are fixed to have a length of 16 characters. In *Leak Set*, the size of secrets are from 12 to 28 bits for *Leak Set 1* to *Leak Set 5*, respectively. Additionally, we apply QFuzz on *Apache WSS4J*, for which we newly reported a vulnerability that has been fixed by the developers [1]. For RQ2, we apply QFuzz on the benchmarks of Blazer, Themis, and DifFuzz. They include micro-benchmarks and real-world Java programs. Finally, for RQ3, we apply QFuzz on the RSA subjects taken from Pasareanu et al. [29]. They represent an unsafe implementation of modular exponentiation, which can leak information about the secret exponent via a timing side channel. We compare our approach to MaxLeak [29] that also quantifies leaks with *min* entropy using symbolic executions and model counting.

[1] https://issues.apache.org/jira/browse/WSS-677

5.2 Technical Details

Our tool QFuzz, similar to DifFuzz [27], is implemented on top of AFL [38] with Kelinci [18] interfacing Java bytecodes. In order to measure the execution cost, the Java programs are instrumented using the ASM bytecode manipulation framework [8]. As a cost metric, we count the Java bytecode instructions, as this metric is substantially used in the related work [1, 9, 27, 29]. This enables a fair comparison and is often sufficient for our analysis of side channels. Generally, the cost metric in QFuzz is easily exchangeable: it supports the execution time, memory consumption, and any other user-specified cost metrics.

DifFuzz stores a cost difference, called *highscore*, internally in the fuzzer and updates it as soon as a mutated input produced a higher cost value. This highscore is calculated as the cost difference between two executions under the same public input with different secret inputs. DifFuzz therefore inherently searches for two partitions and tries to maximize cost differences (δ) between them.

In contrast, QFuzz searches for arbitrary many secret inputs (bound by K) with the same public input. While DifFuzz assesses the inputs by comparing the cost values of two executions, QFuzz assesses the inputs by comparing the cost values of K executions and partitions the costs into classes of *distinguishable* observations. Consequently, QFuzz considers the number of classes (partitions) as a highscore. Additionally, QFuzz maximizes the cost differences (δ) between these partitions as an additional highscore parameter using the minimum distance among the identified partitions.

5.3 Experimental Setup

We used a virtual machine with Ubuntu 18.04.1 *LTS* featuring $2x$ Intel(R) Xeon(R) CPU $X5365$ @ $3.00GHz$ with $8GB$ of memory, OpenJDK 1.8.0_191 and GCC 7.3.0 to run our experiments. Unless noted otherwise, we execute the experiments with a timeout of 30 minutes, a K value of 100, and an ϵ value of 1.0. Each experiment starts with one (randomly generated) seed input, while we only

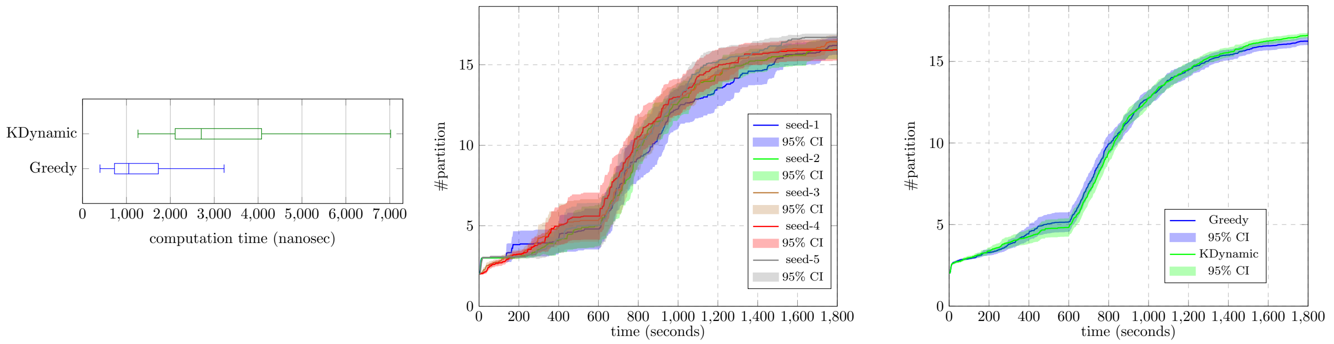

Figure 3: (left) Computational complexity of *Greedy* vs. *KDynamic* in isolation; (middle) *Eclipse Jetty 1* ($\epsilon = 1$): temporal development of 5 different seed inputs with *Greedy*; (right) *Eclipse Jetty 1* ($\epsilon = 1$): temporal development of *Greedy* and *KDynamic* with 5 different seed inputs combined (lines and bands show averages and 95% confidence intervals across 30 repetitions).

ensured that this initial input does not crash the application, as required by the underlying fuzzing engine. We reuse the same seed inputs for comparison with *Greedy* and *KDynamic*, so that both start at the same initial state. The experiments for RQ1 have been repeated with 5 different seed inputs to observe the different behavior of our fuzzer for different initial inputs. For the other experiments (RQ2 and RQ3), we only report the results for one seed input because preliminary experiments as well as the experiments for RQ1 showed that there is no significant variance in the behavior. Additionally, each experiment has been repeated 30 times in order to incorporate the randomness of our fuzzing approach. We averaged the results and calculated the 95% confidence intervals as well as the maximum/minimum values.

5.4 Partitioning Algorithms (RQ1)

In Section 4, we propose two partitioning strategies: *KDynamic* and *Greedy*. The partitioning is a part of our mutant fitness evaluation during fuzzing, which should be as efficient as possible to not slow down the overall fuzzing process. From a theoretical perspective, *KDynamic* incorporates the maximization of δ, while *Greedy* simply tries to find actual partitions. Therefore, *KDynamic* is designed to produce high δ values, while *Greedy* is designed to be fast. In a preliminary experiment we compared *Greedy* and *KDynamic* in isolation, i.e., just the partitioning for $n = 4779$ input files. We extract the public value and $K = 100$ secret values from the input files and then measure the execution time, which each partitioning algorithm needs to calculate the number of partitions. Figure 3 (left) shows the statistical comparison: *KDynamic* is (in isolation) significantly slower and the mean execution time is 1.6 times larger. Please note that the measured time is in nanoseconds, so that the absolute differences between the two execution times might not be essential for the surrounding fuzzing process.

In order to see whether there is a observable difference in the surrounding fuzzing process, we performed additional experiments as described in Section 5.1 with regard to the number of identified partitions, the cost differences δ, and time to the maximum number of partitions. Table 1 shows the results of these experiments. The column $\#Partitions$ shows the true number of partitions, which should be identified for these subjects. The columns $\overline{p}$ and p_{max} describe the average number of partitions with the 95% confidence interval and the maximum number of partitions over 30 runs, respectively. The column δ_{max} describes the δ value for the p_{max}. The column $Time(s) : \overline{p} > 1$ shows the average time to identify more than one partition (with the 95% confidence interval), $Time(s) : p_{max}$ shows the average time to the p_{max} value and $t^{min}_{p_{max}}$ shows the minimum time to find the p_{max} over all runs.

Overall, there was no significant difference in the number of partitions identified by both algorithms. There have been minor discrepancies in favor of the *Greedy* algorithm (as highlighted in red). Manual inspections showed that *KDynamic* algorithm over-approximates the number of partitions (by one partition) for *Eclipse Jetty 4* ($\epsilon = 4$). The differences in the δ values are minor or due to different p_{max} values. For some cases even *Greedy* achieved better δ values, e.g., *Eclipse Jetty 4* ($\epsilon = 4$). The differences between the computation times of *Greedy* and *KDynamic* are mostly insignificant or due to different p_{max} values. We therefore conclude that the measured time difference in the preliminary experiments is too small to affect the overall fuzzing results. The overall fuzzing process with mutations, I/O operations etc. takes longer and outweighs the partitioning effort.

In addition to assessing the final results, we also compared the temporal development of the identified partitions and δ values between them over five different seed inputs. Figure 3 (middle+right) shows this exemplary for the subject *Eclipse Jetty 1* with $\epsilon = 1$ (the plots for the other subjects can be found in the collected experimental results on our GitHub repository). We found that there is no significant difference between the 5 different seed inputs over all subjects, and the two partitioning algorithms perform very similar during the 30 minutes experiments.

Answer RQ1: There is no significant difference in terms of the number of partitions or computation time (during fuzzing) between *KDynamic* and *Greedy*. Interestingly, *Greedy* did produce comparable δ values. We choose *Greedy* as the partition algorithm for the our remaining experiments because it provides an under-approximation and is more efficient in isolation.

Table 1: Comparison of partitioning algorithms (discrepancies are highlighted in *red*).

Subject	ϵ	#Partitions	Algorithm	$\overline{p}$	p_{max}	δ_{max}	Time (s) $\overline{p} > 1$	Time (s) p_{max}	Time (s) $t^{min}_{p_{max}}$
Eclipse Jetty 1	1	17	Greedy	16.24 (+/- 0.24)	17	3	4.65 (+/- 0.10)	1436.16 (+/- 53.58)	675
			KDynamic	16.59 (+/- 0.15)	17	3	4.54 (+/- 0.10)	1423.65 (+/- 51.89)	656
Eclipse Jetty 1	4	9	Greedy	8.63 (+/- 0.10)	9	6	31.20 (+/- 6.38)	1428.63 (+/- 56.03)	697
			KDynamic	8.65 (+/- 0.12)	9	3	31.71 (+/- 6.53)	1394.75 (+/- 55.87)	669
Eclipse Jetty 4	1	9	Greedy	8.51 (+/- 0.09)	9	1	3.93 (+/- 0.11)	1426.89 (+/- 68.86)	496
			KDynamic	8.45 (+/- 0.11)	9	1	4.15 (+/- 0.11)	1437.27 (+/- 66.16)	497
Eclipse Jetty 4	4	4	Greedy	3.94 (+/- 0.04)	*4*	5	28.50 (+/- 4.61)	1083.39 (+/- 59.33)	245
			KDynamic	4.27 (+/- 0.08)	*5*	2	27.21 (+/- 4.07)	1678.88 (+/- 38.36)	599
Eclipse Jetty 5	1	1	Greedy	1.00 (+/- 0.00)	1	0	-	1.00 (+/- 0.00)	1
			KDynamic	1.00 (+/- 0.00)	1	0	-	1.00 (+/- 0.00)	1
Eclipse Jetty 5	4	1	Greedy	1.00 (+/- 0.00)	1	0	-	1.00 (+/- 0.00)	1
			KDynamic	1.00 (+/- 0.00)	1	0	-	1.00 (+/- 0.00)	1
Leak Set 1	1	13	Greedy	13.00 (+/- 0.00)	13	92	4.20 (+/- 0.07)	77.53 (+/- 8.31)	8
			KDynamic	13.00 (+/- 0.00)	13	92	3.77 (+/- 0.09)	79.77 (+/- 9.36)	8
Leak Set 2	1	17	Greedy	17.00 (+/- 0.00)	17	92	4.13 (+/- 0.07)	389.46 (+/- 30.03)	93
			KDynamic	16.99 (+/- 0.01)	17	92	3.73 (+/- 0.12)	379.14 (+/- 35.43)	42
Leak Set 3	1	21	Greedy	20.92 (+/- 0.04)	21	92	5.01 (+/- 0.01)	815.82 (+/- 72.93)	223
			KDynamic	20.89 (+/- 0.05)	21	92	5.00 (+/- 0.00)	842.37 (+/- 78.11)	208
Leak Set 4	1	25	Greedy	24.39 (+/- 0.12)	25	92	5.00 (+/- 0.00)	1437.27 (+/- 72.39)	453
			KDynamic	24.50 (+/- 0.11)	25	92	3.56 (+/- 0.13)	1389.97 (+/- 71.79)	268
Leak Set 5	1	29	Greedy	27.71 (+/- 0.17)	29	92	4.34 (+/- 0.08)	1668.63 (+/- 46.38)	405
			KDynamic	27.66 (+/- 0.16)	29	92	3.77 (+/- 0.11)	1689.11 (+/- 44.18)	426
Apache WSS4J	1	17	Greedy	12.70 (+/- 0.40)	17	3	4.93 (+/- 0.13)	1772.50 (+/- 15.58)	1079
			KDynamic	12.72 (+/- 0.41)	17	3	4.89 (+/- 0.13)	1772.54 (+/- 14.05)	1301

Table 2: The results of applying QFuzz to the Blazer benchmarks (discrepancies are highlighted in *red*).

Benchmark	Version	QFuzz p_{max}	QFuzz δ_{max}	DifFuzz δ_{max}	Time (s) QFuzz, $\overline{p} > 1$	Time (s) DifFuzz, $\overline{\delta} > 0$	Time (s) Blazer	Time (s) Themis
Array	Safe	1	0	1	-	7.40 (+/- 1.21)	1.60	0.28
Array	Unsafe	2	192	195	5.70 (+/- 0.21)	7.40 (+/- 0.93)	0.16	0.23
LoopAndbranch	*Safe*	*2*	*4*	*4,278,268,702*	*1045.33 (+/- 43.51)*	*18.60 (+/- 6.40)*	*0.23*	*0.33*
LoopAndbranch	Unsafe	2	4	4,294,838,782	1078.63 (+/- 61.04)	10.60 (+/- 2.62)	0.65	0.16
Sanity	Safe	1	0	0	-	-	0.63	0.41
Sanity	Unsafe	2	3,537,954,539	4,290,510,883	1414.13 (+/- 102.27)	163 (+/- 40.63)	0.30	0.17
Straightline	Safe	1	0	0	-	-	0.21	0.49
Straightline	Unsafe	2	8	8	7.47 (+/- 0.18)	14.60 (+/- 6.53)	22.20	5.30
unixlogin	Safe	-	-	3	-	510 (+/- 91.18)	0.86	-
unixlogin	Unsafe	2	6,400,000,008	3,200,000,008	1784.47 (+/- 21.27)	464.20 (+/- 64.61)	0.77	-
modPow1	Safe	1	0	0	-	-	1.47	0.61
modPow1	Unsafe	22	117	3,068	4.73 (+/- 0.16)	4.80 (+/- 1.11)	218.54	14.16
modPow2	Safe	1	0	9	-	-	1.62	0.75
modPow2	Unsafe	31	1	5,206	294.70 (+/- 104.66)	23.00 (+/- 3.48)	7813.68	141.36
passwordEq	Safe	1	0	0.00	-	-	2.70	1.10
passwordEq	Unsafe	93	2	127	4.57 (+/- 0.22)	8.60 (+/-2.11)	1.30	0.39
k96	Safe	1	0	0	-	-	0.70	0.61
k96	Unsafe	93	2	3,087,339	4.57 (+/- 0.22)	3.40 (+/- 0.98)	1.29	0.54
gpt14	*Safe*	*12*	*1*	*517*	*5.00 (+/- 0.00)*	*4.20 (+/- 0.80)*	*1.43*	*0.46*
gpt14	Unsafe	92	2	12,965,890	5.87 (+/- 0.12)	4.40 (+/- 1.03)	219.30	1.25
login	Safe	1	0	0	-	-	1.77	0.54
login	Unsafe	17	2	62	7.77 (+/- 0.69)	10.00 (+/- 2.92)	1.79	0.70

5.5 Comparison with Blazer, Themis, and DifFuzz (RQ2)

QFuzz is useful for both the detection and quantification of side channels (SC), and hence, we can compare QFuzz to detection techniques. We compare QFuzz with Blazer [1], Themis [9], and DifFuzz [27], the three state-of-the-art SC detectors. Furthermore, QFuzz's implementation is an extension of DifFuzz, and therefore can serve as a baseline with regard to side-channel detection.

The benchmark of the related studies include subjects mostly in two variants: *safe* and *unsafe*. A *safe* variant is supposed to not show any side-channel vulnerability, while the *unsafe* variant is known to include a vulnerability. For *unsafe* subjects, QFuzz should identify at least 2 partitions. This indicates some measurable differences in the public observations depend on the secret values. Consequently, for *safe* subjects, QFuzz should identify exactly one partition.

Tables 2, 3, and 4 show the corresponding results. The columns for QFuzz only show the p_{max} and δ_{max} since these parameters are the most relevant ones in detecting side channels. In order to compare the analysis time, we report the time until QFuzz identifies at least two partitions, which compares well with DifFuzz's time parameter $\delta > 0$. Our default value of $K = 100$ was not applicable to a few subjects: *LoopAndBranch*, *Sanity*, and *UnixLogin* because they represent relatively expensive executions. As mentioned in Section 4, the fuzzer performs K concrete executions to collect the observations. Since a large value of K strongly influences the time

Table 3: The results of applying QFuzz to the Themis benchmarks (discrepancies are highlighted in *red*).

Benchmark	Version	QFuzz			DifFuzz		Themis		
		p_{max}	δ_{max}	Time (s): $\overline{p} > 1$	δ_{max}	Time (s): $\overline{\delta} > 0$	$\epsilon = 64$	$\epsilon = 0$	Time (s)
Spring-Security	Safe	1	0	-	1	9.00 (+/- 1.26)	✓	✓	1.70
Spring-Security	Unsafe	2	149	13.07 (+/- 0.91)	149	8.80 (+/- 1.16)	✓	✓	1.09
JDK7-MsgDigest	Safe	1	0	-	1	15.80 (+/- 3.93)	✓	✓	1.27
JDK6-MsgDigest	Unsafe	2	239	4.97 (+/- 0.24)	34,479	7.40 (+/- 1.29)	✓	✓	1.33
Picketbox	Safe	1	0	-	1	29.20 (+/-5.00)	✓	✗	1.79
Picketbox	Unsafe	16	2	4.20 (+/- 0.14)	8,794	16.80 (+/- 2.58)	✓	✓	1.55
Tomcat	Safe	3	2	1.93 (+/- 0.13)	14	13.80 (+/- 1.29)	✓	✗	9.93
Tomcat	*Unsafe*	*3*	*2*	*1.97 (+/- 0.11)*	*37*	*128.60 (+/- 87.20)*	✓	✓	*8.64*
Jetty	*Safe*	*31*	*1*	*4.23 (+/- 0.15)*	*8898*	*9.40 (+/- 1.86)*	✓	✓	*2.50*
Jetty	Unsafe	17	2	4.27 (+/- 0.21)	16020	7.00 (+/- 1.05)	✓	✓	2.07
orientdb	Safe	1	0	-	6	3.20 (+/- 0.97)	✓	✗	37.99
orientdb	Unsafe	17	2	1.00 (+/- 0.00)	19,300	3.00 (+/- 0.84)	✓	✓	38.09
pac4j	Safe	2	10	2.00 (+/- 0.00)	10	5.00 (+/- 1.22)	✓	✗	3.97
pac4j	*Unsafe*	*2*	*11*	*2.00 (+/- 0.00)*	*11*	*8.00 (+/- 2.76)*	✓	✓	*1.85*
pac4j	*Unsafe**	*2*	*39*	*2.03 (+/- 0.06)*	*39*	*10.80 (+/- 5.80)*	-	-	-
boot-auth	Safe	2	5	0.90 (+/- 0.11)	5	5.20 (+/- 0.20)	✓	✗	9.12
boot-auth	Unsafe	33	3	0.93 (+/- 0.09)	101	5.20 (+/- 0.20)	✓	✓	8.31
tourPlanner	Safe	1	0	-	0	-	✓	✓	22.22
tourPlanner	Unsafe	51	1	19.97 (+/- 0.24)	576	19.20 (+/- 0.80)	✓	✓	22.01
DynaTable	Unsafe	18	2	5.83 (+/- 0.13)	97	3.60 (+/- 1.21)	✓	✓	1.165
Advanced_table	Unsafe	2	93	11.13 (+/- 1.37)	97	11.20 (+/- 1.62)	✓	✓	2.01
OpenMRS	Unsafe	2	206	629.43 (+/- 10.41)	206	11.60 (+/- 3.22)	✓	✓	9.71
OACC	*Unsafe*	*18*	*2*	*0.97 (+/- 0.06)*	*47*	*7.00 (+/- 1.30)*	✓	✓	*1.83*

Table 4: Results on DifFuzz's additional examples.

Benchmark	Version	QFuzz			DifFuzz	
		p_{max}	δ_{max}	Time (s): $\overline{p} > 1$	δ_{max}	Time (s): $\overline{\delta} > 0$
CRIME	unsafe	33	1	5.00 (+/- 0.00)	782	7.40 (+/- 1.12)
ibasys (imageMacher)	unsafe	9	9	48.30 (+/- 8.21)	262	6.20 (+/- 0.66)
Apache ftpserver Clear	safe	1	0	-	1	7.20 (+/- 1.24)
Apache ftpserver Clear	unsafe	17	2	0.87 (+/- 0.12)	47	6.80 (+/- 1.07)
Apache ftpserver MD5	safe	1	0	-	1	4.20 (+/- 1.93)
Apache ftpserver MD5	unsafe	5	9	0.90 (+/- 0.11)	151	2.80 (+/- 1.11)
Apache ftpserver SaltedPW	(safe)	43	1	12.10 (+/- 0.19)	198	2.20 (+/- 0.73)
Apache ftpserver SaltedPW	unsafe	42	1	11.93 (+/- 0.18)	193	3.60 (+/- 1.08)
Apache ftpserver SaltedPW*	unsafe	54	1	11.90 (+/- 0.17)	178	5.40 (+/- 0.98)
Apache ftpserver StringUtils	safe	1	0	-	0	-
Apache ftpserver StringUtils	unsafe	17	3	4.27 (+/- 0.16)	53	3.00 (+/- 1.05)
AuthMeReloaded	safe	1	0	-	1	7.60 (+/- 0.75)
AuthMeReloaded	unsafe	5	3	2.00 (+/- 0.00)	383	9.20 (+/- 1.96)

for input assessments, we ultimately used $K = 20$ for *LoopAndBranch*, $K = 10$ for *Sanity*, and $K = 2$ for *UnixLogin*, which are sufficient for detecting side channels.

5.5.1 Results with regard to Partitions (RQ2/part1). In all unsafe subjects, QFuzz identifies at least 2 partitions, for which DifFuzz also identifies a high δ value. There are discrepancies with Blazer and Themis classifications/results (see red highlights in Table 2 and 3), but these have been already reported with DifFuzz in [27] and essentially represent false classifications by Blazer and Themis.

QFuzz's capabilities go beyond identifying side-channel vulnerabilities. For example, Blazer's *login* subjects (see Table 2) handle the checking of a secret password (in our case with a fixed length of 16 characters). Our technique identifies 17 partitions with a minimum cost difference of two bytecodes between partitions ($\delta = 2$). This represents the partitions ranging from the 0 (no) prefix character matching to all 16 prefix characters matching for a given public input. Therefore, QFuzz gives stronger evidence that this vulnerability is actually exploitable. Such a statement cannot be inferred with the results from DifFuzz because it only provides the maximum δ value between any two partitions.

Another example is the *unsafe* variant of *Spring-Security* in the Themis benchmark, which had been already mentioned in Section 2. DifFuzz, Themis, and QFuzz conclude that it is unsafe. However, the provided information to assess the severity of this subject is quite different. In this subject, the secret password also has a fixed length of 16 characters. The String comparison is very similar to the safe variant of *Eclipse Jetty* (see Figure 2). The fuzzing driver ensures that public and secrets Strings have the same length, so that the comparison is based on the content of the Strings. Hence, one would expected that no cost differences are possible. But, there is an additional length-check in the beginning: the *expected* lengths of both Strings are compared based on the `String.getBytes("UTF-8")` function. There is an early return, if the lengths of the resulting byte arrays do not match.

While DifFuzz reports a cost difference of 149 bytecodes, QFuzz reports that there are only 2 partitions that are 149 bytcodes far from each other. Therefore, DifFuzz indicates some vulnerability, whereas QFuzz shows that the strength of the leak is weak, and the vulnerability is unlikely to leak the complete secret. A closer look into the identified partitions reveals that the side channels only

Table 5: The results of applying QFuzz to the RSA subjects in MaxLeak [29] (red highlighted exceeded the budget: timeout of 1 hour or memory of 8GB, *blue* highlighted partitions are below the maximum possible observation).

Modulo	Len	#Partitions	QFuzz (ϵ=0, 1h)				MaxLeak (default)		MaxLeak (No solver)	
			$\overline{p}$	p_{max}	Time (s): p_{max}	t_{min}	#Obs	Time (s)	#Obs	Time (s)
1717	3	7	7.00 (+/- 0.00)	7	1.00 (+/- 0.00)	1	6	20.892	9	1.047
1717	4	10	10.00 (+/- 0.00)	10	7.43 (+/- 0.45)	5	9	152.332	12	1.370
1717	5	13	13.00 (+/- 0.00)	13	20.40 (+/- 3.87)	6	12	839.788	15	2.916
1717	6	16	16.00 (+/- 0.00)	16	294.60 (+/- 53.17)	22	15	**3731.328**	18	8.006
1717	7	19	18.37 (+/- 0.25)	19	2484.30 (+/- 451.42)	385		**> 4 h**	21	19.241
1717	8	22	20.43 (+/- 0.45)	22	3168.07 (+/- 303.47)	508		**> 4 h**	24	91.821
1717	9	25	22.20 (+/- 0.36)	*24*	3489.03 (+/- 169.19)	1009		**> 4 h**		**> 8 GB**
1717	10	28	24.40 (+/- 0.49)	*27*	3548.63 (+/- 57.73)	2929		**> 4 h**		**> 8 GB**
834443	3	7	7.00 (+/- 0.00)	7	13.40 (+/- 1.96)	8	6	7.416	9	1.188
834443	4	10	10.00 (+/- 0.00)	10	40.33 (+/- 12.14)	6	9	42.684	12	1.385
834443	5	13	12.93 (+/- 0.09)	13	645.70 (+/- 329.43)	74	12	215.929	15	2.953
834443	6	16	15.40 (+/- 0.20)	16	2711.87 (+/- 433.23)	271	15	936.921	18	7.511
834443	7	19	16.80 (+/- 0.33)	18	3227.60 (+/- 275.29)	952	18	**4021.150**	21	19.068
834443	8	22	17.93 (+/- 0.54)	22	3556.70 (+/- 83.44)	2301		**> 4 h**	24	96.360
834443	9	25	20.13 (+/- 0.59)	*24*	3572.83 (+/- 37.16)	3110		**> 4 h**		**> 8 GB**
834443	10	28	21.83 (+/- 0.46)	*24*	3504.13 (+/- 121.70)	1845		**> 4 h**		**> 8 GB**
1964903306	3	7	6.47 (+/- 0.18)	7	2228.30 (+/- 542.13)	119	6	12.167	9	1.085
1964903306	4	10	8.67 (+/- 0.19)	10	3494.30 (+/- 203.69)	429	9	70.805	12	1.535
1964903306	5	13	10.70 (+/- 0.19)	12	3594.00 (+/- 11.56)	3420	12	2306.261	15	3.391
1964903306	6	16	12.90 (+/- 0.11)	*13*	1337.90 (+/- 443.89)	206		**> 4 h**	18	7.506
1964903306	7	19	14.10 (+/- 0.27)	*15*	2984.67 (+/- 362.05)	503		**> 4 h**	21	19.486
1964903306	8	22	15.33 (+/- 0.36)	*17*	3398.37 (+/- 204.45)	1411		**> 4 h**	24	98.325
1964903306	9	25	16.30 (+/- 0.51)	*19*	3562.33 (+/- 54.24)	2819		**> 4 h**		**> 8 GB**
1964903306	10	28	17.30 (+/- 0.48)	*20*	3559.67 (+/- 77.72)	2390		**> 4 h**		**> 8 GB**

leak whether the secret String contains a special character or not. Although the quantification of information leaks by QFuzz is an under-approximation of the true number of partitions (because of its dynamic nature), it significantly supports the understanding of the vulnerability and the strength of leaks.

Answer RQ2 (Part 1/2): QFuzz detects the same vulnerabilities similar to state-of-the-art techniques. Furthermore, QFuzz provides additional information about the strength of leaks and the exploitability of vulnerabilities.

5.5.2 Results with regard to Analysis Time (RQ2/part2). Comparing the fuzzing time to the first inputs that reveal more than 1 partition, QFuzz is considerably slower than the other techniques in some cases. The large K value in our experiments (usually $K = 100$) triggers a large number of concrete program executions during input assessment in fuzzing. If the program executions are expensive as well, then this can slow down the overall fuzzing campaign. On the one hand, a large value for K enables QFuzz to identify up to K partitions and may lead to a faster exploration via considering multiple secret values. On the other hand, the large value slows down the overall fuzzing process, as the input assessments take longer. The choice of an appropriate value for K remains a trade-off between many partition explorations and a few partition exploitations.

We also observed that in some cases QFuzz is significantly faster than the static analysis techniques Blazer and Themis (e.g., *Straightline unsafe* and *modPow1/2 unsafe*). As reported in Blazer [1], for the long-running benchmarks Blazer suffers from the combinatorial growth of necessary expression comparisons. Themis can improve but still suffers for complex benchmarks. Note that both techniques use taint analysis that is known to be computationally expensive for languages with dynamic features such as Java [23]. QFuzz (as well as DifFuzz) uses a dynamic analysis, which outperforms static analysis in such cases.

Answer RQ2 (Part 2/2): Large values for K may slow down QFuzz, but eventually, enable the exploration of many partitions. QFuzz outperforms static analysis on complex benchmarks.

5.6 Comparison to MaxLeak [29] on RSA subjects (RQ3)

Pasareanu et al. [29] quantify information leaks using symbolic execution and model counting (MaxSMT). In particular, they evaluate their approach on the implementations of fast modular exponentiation. Their cost model does not count executed bytecode instructions, but counts the number of visited branches. To match with their evaluations, we customized our cost model, reduced ϵ to zero, and increased the timeout of the experiments to one hour. To enable a fair comparison, we reproduced MaxLeak's results on our experiment setup.

Table 5 shows the results for these experiments (similar to Figure 9 in [29]). The column *Modulo* denotes the modulo value used for the modulo exponentiation, while the column *Len* denotes the bitvector length of the secret. The column *#Partitions* shows the groundtruth for the number of identifiable partitions. The true number of partitions is formulated to be 3*(*Len*-1) in [29] for every experiment. We noticed that the formulation is for *Len*>1, while we also consider *Len*=1, which leads us to find one more partition. Therefore, we report the *#Partitions* as: 3*(*Len*-1)+1. For QFuzz, we report the average number of identified partitions $\overline{p}$ with the 95% confidence intervals, the maximum number of partitions p_{max}, the average time to p_{max}, and the minimum time to p_{max} over all 30 runs. The results by [29] come in two modes: *default* and *no solver*. The second one represents symbolic execution without filtering

infeasible path constraints, which is faster but over-approximates the number of observations. For each mode, the columns show the number of *different* observations (i.e., number of partitions) and the combined time for symbolic execution and MaxSMT solving.

The results in Table 5 show that for modulo=1717, QFuzz reliably identifies the correct number of partitions up to a bitvector length of 8. For the reported longer secrets, QFuzz identifies a close under-approximation within the given timeout of 1 hour. MaxLeak (default) shows longer runtimes and already exceeds the timeout for the bitvector length of 6. Note that we manually stopped the executions after 4 hours (3 hours later than the timeout) so that we cannot report the exact times for longer running subjects. MaxLeak (no solver) is comparably very fast, but over-approximates the right number of partitions. It eventually exceeds our memory budget of 8 GB for bitvector lengths higher or equal to 9. With higher values for *Modulo*, we can still confirm that QFuzz produces the best results. However, it also slows down so that, e.g., for $Modulo = 1964903306$, QFuzz can only produce reliable estimations for bitvectors lengths up to 5. For larger lengths, QFuzz still can provide good lower-bounds for the true number of partitions.

Answer RQ3: Due to its dynamic analysis, QFuzz is more scalable than MaxLeak. We also find that QFuzz with lower-bound guarantees performs well on the RSA subjects and has precision comparable to MaxLeak [29] that uses symbolic execution with model counting.

6 RELATED WORK

Fuzzing and Testing for Side-Channel Detection. Various testing techniques [14, 25, 27] have been employed to identify side channels. We compared our approach to DifFuzz thoroughly in Section 5. Milushev et al. [25] adapt symbolic executions with self-composition techniques [5] to detect the noninterference violations due to direct and indirect information leaks. Recently, He al. [14] adapt greybox fuzzing techniques with self-composition specifications to detect side-channel leaks. Both of these works consider the noninterference notion of security, while our approach also quantifies the amount of information leaks.

Quantitative Analysis of Side Channels. Quantitative information flow [3, 21, 32, 33] has been used for measuring the strength of side channels and mitigating potential leaks. We compared our approach with Pasareanu et al. [29] in Section 5. Smith [32] studies various quantitative notions of confidentiality and finds that they have limitations in evaluating the resulting threats if an attacker can try one ideal guess. Therefore, Smith [32] introduces min entropy and shows its usefulness in quantifying the strength of leaks. Our approach adapts greybox fuzzing with partitioning algorithms to characterize a lower-bound on min entropy. Backes et al. [3] present an approach based on finding the equivalence relation over secret inputs. They cast the problem of finding the equivalence relation as a reachability problem and use model counting to quantify information leaks. Their approach considers only direct information leaks and works for a small program, limited to a few lines of code.

Dynamic Analysis for Side Channels. Dynamic analysis has been used for analyzing side channels [16, 27, 34]. Tizpaz-Niari et al. [34] developed a data-driven approach to debug timing side channels. They define functional side channels where an attacker may observe the response times of application on many public inputs (potentially unbounded). While their approach uses machine learning-based clustering to detect side channels and localize root causes, their clustering is after input generations. Thus, their approach does not guarantee to maximize the number of clusters. In addition, their approach is limited to a setting where an attacker passively observes the response times, whereas we consider a more realistic setting where an attacker can pick public inputs.

Static Analysis for Side Channels. Static analysis techniques are adapted in various works to detect side channels [1, 9, 11, 36]. Themis [9] uses Cartesian Hoare Logic [5] with taint analysis to detect side channels. Similar to DifFuzz, we show that our approach outperforms Themis, and it also characterizes the amounts of leaks.

Adaptive Side Channels. Adaptive side channels allow attackers to use previous observations to pick an ideal public input in each step of an attack [4, 31]. Phan et al. [31] consider synthesizing adaptive side channels where in each step, the attacker chooses the best public input that maximizes the amount of leaks. They reduce the problem of finding k ideal public inputs for compromising secrets to an optimization problem using symbolic execution and MaxSMT [26]. In contrast, we focus on threat models with one ideal public input and adapt greybox fuzzing to characterize min entropy. Thus, Adaptive attacks are beyond the scope of our threat model. Despite it, we show that QFuzz can indicate the possibility of adaptive attacks. We left further analysis as future work.

Side Channels from Micro-Architecture. Recently, significant research has been conducted to address transient execution vulnerabilities such as Spectre attacks [13, 20]. These attacks exploit timing side channels due to speculative executions from micro-architecture sources and can compromise confidentiality even if the source code is completely secure. Our approach focuses on vulnerabilities in the source code since a secure software in the source code level is a precondition to have secure systems. Since the total elimination of Spectre vulnerabilities is not practical (due to performance concerns), our quantitative approach can potentially be useful to guarantee security for micro-architecture designs.

Side Channels from Compiler Optimization. Brennan et al. [6] show potential timing leaks induced from JIT optimizations via actual runtime observations, which is different than our analysis based on vulnerabilities in the source code. We opted to count the bytecode, instead of actual execution times, since the metric is substantially used in the related work for a fair comparison and is often sufficient for our analysis of side channels in code. We note that the cost metric in QFuzz is easily exchangeable: it supports the execution time, bytecode abstraction, memory consumption, and any other user-specified cost metrics.

7 DISCUSSION

Limitations. Our approach requires a driver program, an upper-bound on partitions K, and the tolerance parameter ϵ as inputs. In our evaluation, we discuss the trade-off to pick appropriate values for K. For the tolerance, we usually set $\epsilon = 1$ to be conservative

and comparable to existing works. In practice, developers can run experiments with different ϵ values to adapt for the specific context.

A dynamic analysis, as used in this work, often scales well for large applications and can handle dynamic features. However, such an analysis can lead to false negatives. Nonetheless, QFuzz provides precise under-approximations, and our evaluation shows that QFuzz provides a good indication of resulting threats.

The leveraged min entropy assumes that an attacker can try one time, and it may fail to evaluate the resulting threats if an attacker can try multiple times adaptively. However, we find that a higher number of partitions can serve as warning signs for feasible side-channel attacks over multiple trials such as adaptive attacks.

Threats to Validity. To ensure that our evaluation does not lead to invalid conclusions (i.e., internal validity), we followed the established guidelines [2, 19] to incorporate the randomness in fuzzing. In particular, we repeated all experiments 30 times, showed the error margins, performed experiments with multiple seed inputs, and considered not only the final results but also the temporal development. To illustrate that our results generalize (i.e., external validity), we applied QFuzz on a large range of subjects from existing work including micro- and macro-benchmarks. We specifically used QFuzz to analyze real-world programs and also identified a previously unknown vulnerability. We leveraged evaluation metrics like the number of identified partitions and the time to identify more than one partition because they quantify the information leakage for practical usage and represent how fast our technique can make progress.

Usage Vision. Based on the experiences during our evaluation, we envision the following usage for QFuzz. As a first step, use QFuzz to detect side-channel vulnerabilities, and at the same time quantify their strengths. The obtained quantification facilitates the prioritization of the detected vulnerabilities. Secondly, focus on the manual investigation to mitigate the side channels guided by the results of the first step.

8 CONCLUSION

We presented QFuzz, the first quantitative greybox fuzzing approach for measuring the strength of side-channel leaks. We have shown that QFuzz outperforms existing state-of-the-art techniques for detection and quantification of side-channel vulnerabilities. Furthermore, QFuzz found a zero-day vulnerability in a security-critical Java library that has since been fixed. For future work, we aim to extend QFuzz to quantitatively evaluate the resulting threats from an attacker who can try multiple times adaptively.

Our open-source tool QFuzz and all experimental subjects are publicly accessible:

- https://github.com/yannicnoller/qfuzz
- https://doi.org/10.5281/zenodo.4722965

REFERENCES

[1] Timos Antonopoulos, Paul Gazzillo, Michael Hicks, Eric Koskinen, Tachio Terauchi, and Shiyi Wei. 2017. Decomposition Instead of Self-Composition for Proving the Absence of Timing Channels. *SIGPLAN Not.* 52, 6 (jun 2017), 362–375. https://doi.org/10.1145/3140587.3062378

[2] Andrea Arcuri and Lionel Briand. 2014. A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability* 24, 3 (2014), 219–250. https://doi.org/10.1002/stvr.1486

[3] Michael Backes, Boris Köpf, and Andrey Rybalchenko. 2009. Automatic Discovery and Quantification of Information Leaks. In *2009 30th IEEE Symposium on Security and Privacy*. 141–153. https://doi.org/10.1109/SP.2009.18

[4] Lucas Bang, Abdulbaki Aydin, Quoc-Sang Phan, Corina S Păsăreanu, and Tevfik Bultan. 2016. String Analysis for Side Channels with Segmented Oracles. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 193–204. https://doi.org/10.1145/2950290.2950362

[5] G Barthe, P R D'Argenio, and T Rezk. 2004. Secure information flow by self-composition. In *Proceedings. 17th IEEE Computer Security Foundations Workshop, 2004*. 100–114. https://doi.org/10.1109/CSFW.2004.1310735

[6] Tegan Brennan, Seemanta Saha, and Tevfik Bultan. 2020. JVM Fuzzing for JIT-Induced Side-Channel Detection. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1011–1023. https://doi.org/10.1145/3377811.3380432

[7] Billy Bob Brumley and Nicola Tuveri. 2011. Remote Timing Attacks Are Still Practical. In *Computer Security – ESORICS 2011*, Vijay Atluri and Claudia Diaz (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 355–371. https://doi.org/10.1007/978-3-642-23822-2_20

[8] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. 2002. *ASM: a code manipulation tool to implement adaptable systems.* Technical Report. France Telecom R&D, DTL/ASR. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.117.5769&rep=rep1&type=pdf

[9] Jia Chen, Yu Feng, and Isil Dillig. 2017. Precise Detection of Side-Channel Vulnerabilities Using Quantitative Cartesian Hoare Logic. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. Association for Computing Machinery, New York, NY, USA, 875–890. https://doi.org/10.1145/3133956.3134058

[10] Shuo Chen, Rui Wang, XiaoFeng Wang, and Kehuan Zhang. 2010. Side-Channel Leaks in Web Applications: A Reality Today, a Challenge Tomorrow. In *2010 IEEE Symposium on Security and Privacy*. 191–206. https://doi.org/10.1109/SP.2010.20

[11] Goran Doychev, Boris Köpf, Laurent Mauborgne, and Jan Reineke. 2015. CacheAudit: A Tool for the Static Analysis of Cache Side Channels. *ACM Trans. Inf. Syst. Secur.* 18, 1, Article 4 (June 2015), 32 pages. https://doi.org/10.1145/2756550

[12] Eclipse/GitHub. 2017. Fixed timing side-channel on the length of password in Eclipse Jetty. https://github.com/eclipse/jetty.project/commit/a7e8b4220a410b85c843bffcd13f07d70f1b3fe8

[13] Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. 2020. Spectector: Principled Detection of Speculative Information Flows. In *2020 IEEE Symposium on Security and Privacy (SP)*. 1–19. https://doi.org/10.1109/SP40000.2020.00011

[14] Shaobo He, Michael Emmi, and Gabriela Ciocarlie. 2020. ct-fuzz: Fuzzing for Timing Leaks. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. 466–471. https://doi.org/10.1109/ICST46399.2020.00063

[15] H. V. Jagadish, Nick Koudas, S. Muthukrishnan, Viswanath Poosala, Kenneth C. Sevcik, and Torsten Suel. 1998. Optimal Histograms with Quality Guarantees. In *Proceedings of the 24rd International Conference on Very Large Data Bases (VLDB '98)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 275–286.

[16] İsmet Burak Kadron, Nicolás Rosner, and Tevfik Bultan. 2020. Feedback-Driven Side-Channel Analysis for Networked Applications *(ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 260–271. https://doi.org/10.1145/3395363.3397365

[17] Mehdi Kaytoue, Zainab Assaghir, Amedeo Napoli, and Sergei O. Kuznetsov. 2010. Embedding Tolerance Relations in Formal Concept Analysis: An Application in Information Fusion. In *Proceedings of the 19th ACM International Conference on Information and Knowledge Management* (Toronto, ON, Canada) *(CIKM '10)*. Association for Computing Machinery, New York, NY, USA, 1689–1692. https://doi.org/10.1145/1871437.1871705

[18] Rody Kersten, Kasper Luckow, and Corina S Păsăreanu. 2017. POSTER: AFL-based Fuzzing for Java with Kelinci. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. ACM, New York, NY, USA, 2511–2513. https://doi.org/10.1145/3133956.3138820

[19] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. ACM, New York, NY, USA, 2123–2138. https://doi.org/10.1145/3243734.3243804

[20] P Kocher, J Horn, A Fogh, D Genkin, D Gruss, W Haas, M Hamburg, M Lipp, S Mangard, T Prescher, M Schwarz, and Y Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *2019 IEEE Symposium on Security and Privacy (SP)*. 1–19. https://doi.org/10.1109/SP.2019.00002

[21] Boris Köpf and David Basin. 2007. An Information-Theoretic Model for Adaptive Side-Channel Attacks. In *Proceedings of the 14th ACM Conference on Computer*

and Communications Security (CCS '07). Association for Computing Machinery, New York, NY, USA, 286–296. https://doi.org/10.1145/1315245.1315282

[22] Boris Köpf and Markus Dürmuth. 2009. A Provably Secure and Efficient Countermeasure against Timing Attacks. In *2009 22nd IEEE Computer Security Foundations Symposium*. 324–335. https://doi.org/10.1109/CSF.2009.21

[23] Davy Landman, Alexander Serebrenik, and Jurgen J. Vinju. 2017. Challenges for Static Analysis of Java Reflection - Literature Review and Empirical Study. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 507–518. https://doi.org/10.1109/ICSE.2017.53

[24] Nate Lawson. 2009. Timing attack in Google Keyczar library. https://rdist.root.org/2009/05/28/timing-attack-in-google-keyczar-library/

[25] Dimiter Milushev, Wim Beck, and Dave Clarke. 2012. Noninterference via Symbolic Execution. In *Formal Techniques for Distributed Systems*, Holger Giese and Grigore Rosu (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 152–168. https://doi.org/10.1007/978-3-642-30793-5_10

[26] Robert Nieuwenhuis and Albert Oliveras. 2006. On SAT modulo Theories and Optimization Problems. In *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing* (Seattle, WA) *(SAT'06)*. Springer-Verlag, Berlin, Heidelberg, 156–169. https://doi.org/10.1007/11814948_18

[27] Shirin Nilizadeh, Yannic Noller, and Corina S. Păsăreanu. 2019. DifFuzz: Differential Fuzzing for Side-channel Analysis. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, Piscataway, NJ, USA, 176–187. https://doi.org/10.1109/ICSE.2019.00034

[28] OpenJDK. 2020. Bytecode comparison in OpenJDK. http://hg.openjdk.java.net/jdk8u/jdk8u-dev/jdk/file/26deba50fea8/src/share/classes/java/security/MessageDigest.java#l448

[29] Corina S Pasareanu, Quoc-Sang Phan, and Pasquale Malacaria. 2016. Multi-run Side-Channel Analysis Using Symbolic Execution and Max-SMT. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*. 387–400. https://doi.org/10.1109/CSF.2016.34

[30] James F. Peters and Piotr Wasilewski. 2012. Tolerance Spaces: Origins, Theoretical Aspects and Applications. *Inf. Sci.* 195 (July 2012), 211–225. https://doi.org/10.1016/j.ins.2012.01.023

[31] Q Phan, L Bang, C S Păsăreanu, P Malacaria, and T Bultan. 2017. Synthesis of Adaptive Side-Channel Attacks. In *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*. 328–342. https://doi.org/10.1109/CSF.2017.8

[32] Geoffrey Smith. 2009. On the Foundations of Quantitative Information Flow. In *Foundations of Software Science and Computational Structures*, Luca de Alfaro (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 288–302. https://doi.org/10.1007/978-3-642-00596-1_21

[33] Saeid Tizpaz-Niari, Pavol Černý, and Ashutosh Trivedi. 2019. Quantitative Mitigation of Timing Side Channels. In *Computer Aided Verification*, Isil Dillig and Serdar Tasiran (Eds.). Springer International Publishing, Cham, 140–160. https://doi.org/10.1007/978-3-030-25540-4_8

[34] Saeid Tizpaz-Niari, Pavol Cerný, and Ashutosh Trivedi. 2020. Data-Driven Debugging for Functional Side Channels. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society. https://doi.org/10.14722/ndss.2020.24269

[35] Saeid Tizpaz-Niari and Yannic Noller. 2020. Comparison in validate class of WSS4J Core is vulnerable to timing side channels. https://issues.apache.org/jira/browse/WSS-677

[36] Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. 2017. CacheD: Identifying Cache-Based Timing Channels in Production Software. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 235–252. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/wang-shuai

[37] IVC Wiki. 2007. Xbox 360 Timing Attack. https://beta.ivc.no/wiki/index.php/Xbox_360_Timing_Attack

[38] Michal Zalewski. 2014. American Fuzzy Lop (AFL) - a security-oriented fuzzer. http://lcamtuf.coredump.cx/afl/