DIFFUZZ: Differential Fuzzing for Side-Channel Analysis

Shirin Nilizadeh*
University of Texas at Arlington
Arlington, TX, USA
shirin.nilizadeh@uta.edu

Yannic Noller*
Humboldt-Universität zu Berlin
Berlin, Germany
yannic.noller@hu-berlin.de

Corina S. Păsăreanu
Carnegie Mellon University Silicon Valley,
NASA Ames Research Center
Moffett Field, CA, USA
corina.s.pasareanu@nasa.gov

arXiv:1811.07005v2 [cs.CR] 26 Feb 2019

***Abstract*—Side-channel attacks allow an adversary to uncover secret program data by observing the behavior of a program with respect to a resource, such as execution time, consumed memory or response size. Side-channel vulnerabilities are difficult to reason about as they involve analyzing the correlations between resource usage over multiple program paths. We present DIFFUZZ, a fuzzing-based approach for detecting side-channel vulnerabilities related to time and space. DIFFUZZ automatically detects these vulnerabilities by analyzing two versions of the program and using resource-guided heuristics to find inputs that *maximize* the difference in resource consumption between secret-dependent paths. The methodology of DIFFUZZ is general and can be applied to programs written in any language. For this paper, we present an implementation that targets analysis of JAVA programs, and uses and extends the KELINCI and AFL fuzzers. We evaluate DIFFUZZ on a large number of JAVA programs and demonstrate that it can reveal unknown side-channel vulnerabilities in popular applications. We also show that DIFFUZZ compares favorably against BLAZER and THEMIS, two state-of-the-art analysis tools for finding side-channels in JAVA programs.**

***Index Terms*—vulnerability detection; side-channel analysis; dynamic analysis; fuzzing**

I. INTRODUCTION

Side-channel attacks enable an adversary to uncover sensitive information from programs by observing non-functional characteristics of program behavior, such as execution time, memory usage, response size, network traffic, or power consumption. There is a large literature on side channels showing evidence that they are practical and can have serious security consequences [13], [22], [31]. For instance, exploitable timing channel information flows were discovered for Google's Keyczar Library [28], the Xbox 360 [5] and implementations of RSA encryption [13]. More recently, the Meltdown and Spectre side-channel attacks [4] have shown how to exploit critical vulnerabilities in modern processors to uncover secret information. These vulnerabilities highlight the increased need for tools and techniques that can effectively discover side channels before they are exploited by a malicious user in the field. However, side-channel vulnerabilities are difficult to reason about as they involve analyzing correlations between resource usage over multiple program paths.

In this paper we present DIFFUZZ, a dynamic analysis approach for the detection of side channels in software systems. Given a program whose inputs are partitioned into public and secret variables, DIFFUZZ uses a form of differential fuzzing to automatically find program inputs that reveal side channels related to a specified resource, such as time, consumed memory, or response size. We focus specifically on timing and space related vulnerabilities, but the approach can be adapted to other types of side channels, including cache based.

Differential fuzzing has been successfully applied before for finding bugs and vulnerabilities in a variety of applications, such as LF and XZ parsers, PDF viewers, SSL/TLS libraries, and C compilers [36], [38], [41]. However, to the best of our knowledge, we are the first to explore differential fuzzing for side-channel analysis. Typically such fuzzing techniques analyze different versions of a program, attempting to discover bugs by observing differences in execution for the same inputs. In contrast DIFFUZZ works by analyzing two copies of the same program, with the same public inputs but with different secret values, and computing the *difference* in side channel measurements (time or space) observed over the two executions. If the difference is large, then it means that the program has a side-channel vulnerability, which should be remedied by the developer.

The approach is similar to the well-known method of *self-composition* [10]), which is used to check that no matter what the secret is, the program yields the same output. If that is the case, the program is said to satisfy *non-interference*, meaning that the program leaks *no* information; otherwise, the program is vulnerable. However, it has been argued [8], [14] that non-interference with regard to side channels is too strong a property for most realistic programs, as it is almost always the case that some variation in resource usage, particularly execution time, exists for different program paths. If the difference is small, it may not be exploitable in practice, since it may not be actually observable by an attacker. For example, consider the case of a client-server application. Small variations in execution time on the server side may not be observable (and therefore exploitable) on the client side. In such cases the program can be considered secure although it does not satisfy non-interference. If on the other hand, the difference is large, this indicates a side-channel vulnerability since an attacker can use differences between measurements to distinguish between secrets. For this reason, DIFFUZZ does not merely check non-interference, but instead employs

*Joint first authors

resource-guided heuristics to automatically find inputs that attempt to *maximize* the difference in resource consumption between secret-dependent paths.

The methodology that we advocate with DIFFUZZ is general and can be applied to programs written in any language. For this paper we present an implementation that targets JAVA programs and is based on AMERICAN FUZZY LOP (AFL) [42] and KELINCI [24]. AFL is a fuzz testing tool that uses genetic algorithms to mutate user-provided inputs using byte-level operations with the goal of increasing coverage; KELINCI provides an interface to execute AFL on JAVA programs. To perform side-channel analysis, DIFFUZZ instruments a program to record resource consumption, in addition to coverage, along the paths that are executed by the fuzzed inputs. Furthermore, DIFFUZZ records the *difference* in resource consumption in a user-defined cost. This difference is sent back to the fuzzer, whose mutants are marked as *important* if there is an increase in the computed difference, thus *guiding* the fuzzer towards inputs that expose vulnerabilities.

We have applied DIFFUZZ on well-known, widely used JAVA applications, such as Apache FtpServer [1] and AuthMeReloaded [2], where we found new, previously unknown, vulnerabilities, which were confirmed by the developers. Additionally we have applied our approach on complex examples from the DARPA Space/Time Analysis for Cybersecurity (STAC) program [20], IBASys, an image-based authentication system, and CRIME, an instance of the Compression Ratio Info-leak Made Easy attack [19], where we found vulnerabilities related to both time and space consumption.

We also compared DIFFUZZ with BLAZER [8] and THEMIS [14], two state-of-the-art analysis tools for finding side channels in JAVA programs. Both tools perform static analysis and can in principle guarantee absence of side channels, but may also give false alarms due to underlying over-approximation. In contrast DIFFUZZ performs a dynamic analysis, and thus does not give false alarms (provided that the fuzzing driver is meaningful, see Section 2.3.), but it can not prove absence of vulnerabilities.

We evaluated DIFFUZZ on the same benchmarks from THEMIS and BLAZER and were able to find the same vulnerabilities. We also ran DIFFUZZ on the corrected (safe) versions (when they were available). For the majority of these cases, we found that as expected, DIFFUZZ correctly finds zero or a small differences thus showing the usefulness of the approach also in the case of absence of vulnerabilities. However, we have also found that, in some cases, DIFFUZZ uncovered new vulnerabilities in versions which were shown to be safe with BLAZER and THEMIS.

In summary, this work makes the following contributions:

- We present DIFFUZZ, the first differential fuzzing approach for finding side-channel vulnerabilities.
- We evaluate DIFFUZZ on multiple security-critical JAVA applications and we report new vulnerabilities in well known JAVA applications, such as Apache FtpServer.
- We compare with state-of-the-art tools BLAZER and THEMIS, where we highlight some new vulnerabilities in programs that were previously deemed safe.

II. APPROACH

Figure 1 shows the overview of our differential fuzzing approach. To start the analysis, the user needs to provide initial seed files that exercise the program under test (cf. step 1 in Figure 1). The user also needs to provide a *driver*, which *parses* an input file into three elements pub (common public value), sec_1, and sec_2 (two secret values, one for each program copy) and executes two copies of the program on these inputs. The program is *instrumented* to record resource consumption and coverage information.

The seed files are put into a queue for further processing (cf. step 2 in Figure 1). This queue is used during the whole process as the central data structure that includes all the inputs that are deemed *interesting* by the analysis. The fuzzer will take the inputs from the queue and will mutate them repeatedly (cf. step 3 in Figure 1). In order to decide whether a mutated input is interesting for further processing, DIFFUZZ executes the driver with this input, computes the cost difference between two executions, which is handled as the score for this input, and compares it with the maximum cost difference (aka cost difference *high-score*), which was observed in the previous executions (cf. step 4 in Figure 1).

Only the inputs that either lead to increased high-score or to increased overall program coverage will be forwarded to the fuzzing queue (cf. step 5 in Figure 1). The process is repeated until a user-specified timeout occurs.

We describe the DIFFUZZ approach in more detail below.

A. Side-Channel Analysis

Information flow analysis is typically used to determine that a program manipulates secret data in a secure manner. The analysis accepts programs as secure if the secret data can not be inferred by an attacker through their observations of the systems. This intuitive property is called *non-interference*. In the case of side-channels, the *observations* consist of the side-channel measurements that an attacker can make.

There are many techniques for checking non-interference. The simplest one is through *self-composition* [10]. At a high level the technique reduces the problem of secure information flow of a program to analyzing two copies of the same program, where the secret inputs are renamed, but the public values stay the same, and checking that these two copies create the same observation.

Let P be a program, and $P[\![pub, sec]\!]$ be the execution of the program P with inputs pub and sec. As it is customary in the security literature, we break down the program inputs to a tuple of public (low) values and secret (high) values. We abbreviate the public values as pub and the secret values as sec. Furthermore let $c(.)$ be the evaluation of a program execution with respect to a particular cost encoding the resource usage (e.g., execution time or response size) of the program. The non-interference requirement can then be formalized as follows:

$$\forall pub, sec_1, sec_2 : c(P[\![pub, sec_1]\!]) = c(P[\![pub, sec_2]\!])$$

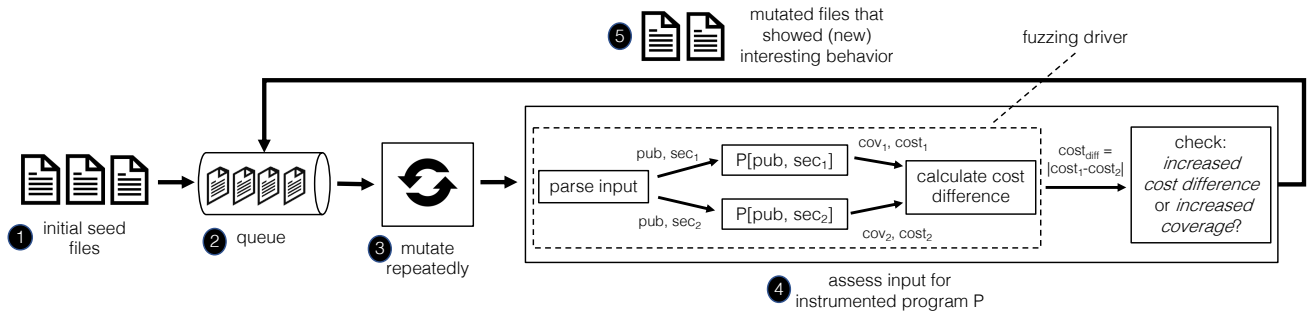

Fig. 1: Overview of DIFFUZZ approach.

Intuitively, the property states that any two secrets are *indistinguishable* through the side-channel observations and therefore can not be uncovered by an attacker.

Although satisfying non-interference is a sound guarantee for a system to be secure, this requirement is too strict for the side-channel analysis of most realistic programs. Particularly for timing channels, small differences in computations may be imperceptible to an attacker and can thus not be exploited in practice. This problem was observed in various papers before [8], [14] and was formalized as checking ϵ-bounded non-interference in [14]: not only programs with zero interference can be accepted as secure, but also programs where the difference between observations is too small (below a threshold ϵ) to be exploitable in practice. Thus the program is deemed to be secure if the following condition holds:

$$\forall pub, sec_1, sec_2 : |c(P[\![pub, sec_1]\!]) - c(P[\![pub, sec_2]\!])| < \epsilon$$

One can perform the above check by enumerating all the possible input combinations, measuring the resource consumption for each run, and performing the check for the two versions of the program, but this could become quickly intractable for most realistic programs.

We therefore advocate the use of fuzzing to address the problem. However typical fuzzing tools are engineered to only increase code coverage and can thus be very slow in generating inputs that expose a significant difference in resource consumption. The key ingredient of our approach is the incorporation of heuristics that guide the fuzzing towards configurations that maximize this difference, as explained in the following sections. Note that, unlike previous techniques, that use static analysis to check ϵ-bounded non-interference [8], [14], we do not require the user to provide an a-priori threshold ϵ; instead we let the tool try to maximize the difference between secret-dependent paths.

B. Attacker Model

We review here the attacker model considered in this paper, which is similar to previous work on the topic [8], [14]. We assume the program is deterministic and that the side-channel measurements are precise. We further assume that the attacker can not observe anything else (i.e., the attacker does not use the main-channel to infer information). When measuring resource usage we assume that any variations are caused by the application software, and we are thus ignoring side-channels related to the hardware architecture or the physical environment. In principle we can handle all these side-channels by using an available model of the corresponding resource. Even in the absence of a model, we could use the inputs generated by the fuzzer to run the programs on a specific platform and perform actual, precise measurements with respect to the resource of interest. Furthermore, we could measure the wall-clock time and also the JIT (just-in-time compilation) effect.

The mentioned assumptions are realistic. For example imagine a server-client scenario in a distributed environment (similar described in [14]), in which the attacker is physically separated from the victim application, i.e. there is no chance to observe any physical side-channel. For an encrypted network communication the attacker cannot read the content of the sent messages, and hence, relies on the metrics that can be observed during communication with the server, like response sizes and response times. Additionally, based on the physical distribution the attacker should not have the possibility to manipulate the victim application to observe any hardware-level side-channels.

Note that DIFFUZZ is also applicable to non-deterministic code and in the experiments we report on such an application. However, in general the results could be imprecise in this case, due to the *noise* introduced in the measurements. More analysis would be necessary, which is left for future work.

C. Differential Fuzzing

Our approach aims to use fuzzing to analyze the two copies of the program and to *guide* it to find inputs that maximize the cost difference between two program executions, for which only the secret values are different:

$$\underset{pub, sec_1, sec_2}{\text{maximize:}}\ \delta = |c(P[\![pub, sec_1]\!]) - c(P[\![pub, sec_2]\!])| \quad (1)$$

Fuzzing Driver*:* In order to apply fuzzing we need a *driver* that parses the inputs from the fuzzer and executes the two copies of the code under test, while also measuring the cost difference. Procedure 1 shows the general driver of our fuzzing approach. It starts with parsing the input (cf. line 1), i.e., reading three different input values: the public value and two secret values, which are used to execute the program twice, as formulated in Equation 1. Additionally, the parsing can take

some simple constraints for these input values, as described below. For each execution we measure the costs (cf. line 2 and 3) and calculate the absolute cost difference (cf. line 4). This value is used to guide the fuzzer (cf. line 5) to generate more inputs with the goal of increasing the difference. In our implementation this notion of cost difference is realized by setting user-defined cost values.

Procedure 1 Differential Fuzzing Driver

1: $pub, sec_1, sec_2 \leftarrow$ parse$(input, constraints)$
2: $cost_1 \leftarrow$ measure$(P(pub, sec_1))$
3: $cost_2 \leftarrow$ measure$(P(pub, sec_2))$
4: $cost_{diff} \leftarrow |cost_1 - cost_2|$
5: setUserDefinedCost$(cost_{diff})$

Input Constraints*:* Solving the maximization problem described in Equation 1 for two totally arbitrary chosen input tuples might not be expedient because most applications assume certain properties of the secret values. For example if a password is stored as a hash, the application would assume that the hashed values have the same fixed length. Using secret values with arbitrary lengths for testing this application would lead to results that are not meaningful. Therefore, in practice it is useful to set some simple constraints on the inputs. In our approach we have a constructive solution, i.e. we rely on the user to encode input constraints in the driver such that only the inputs that satisfy these constraints are passed to the programs. For example the driver can limit the size of a string during parsing by simply not reading more characters than a given threshold, or the driver can ensure a certain character set for a string that should represent a hash value by mapping all non-member characters to member characters during parsing.

Analysis Outcome*:* The result of the analysis is a set of concrete public and secret inputs that expose the maximum cost difference between two secret-dependent paths found by the fuzzer. If the difference is large, it indicates a side-channel vulnerability and the developer can use the provided inputs to precisely pinpoint the problem and fix the vulnerability, e.g., by making the cost similar on both program paths. If on the other hand the difference is small (or zero) it could mean that the program has no vulnerabilities or that the fuzzer was not run long enough. The developer can then run the fuzzer longer to get enough confidence that the software indeed has no vulnerability. The fuzzer also records the *coverage* achieved on the analyzed code and this information can also be examined to increase the confidence in the reported results.

Manual Effort*:* DIFFUZZ requires manual effort in writing the drivers and the input constraints. In the driver, the user needs to specify: how to parse the input file to retrieve valid input values, the entry point to start the target application, and how to measure the execution cost. As many applications come with test cases, we use them to determine entry points. We believe that the manual effort is not high as all the drivers are very similar (and follow Procedure 1); the constraints are minimal and application specific (e.g., passwords have certain lengths). One can also envision using fuzzing to discover these constraints automatically, following the work on grammar inference from [21]. However this is left for future work.

D. Fuzzing Programs

For fuzzing we use off-the-shelf tools such as AFL [42]. AFL is a state-of-the-art, security-oriented grey-box fuzzer that employs compile-time instrumentation and genetic algorithms to automatically generate test inputs that improve the branch coverage of the analyzed code.

Fuzz testing tools have been very successful at finding bugs and vulnerabilities in a variety of applications, ranging from image processors and web browsers to system libraries and various language interpreters. For example, AFL was instrumental in finding several of the Stagefright vulnerabilities in Android, the Shellshock related vulnerabilities, vulnerabilities in BIND, as well as numerous bugs in (security-critical) applications and libraries such as OPENSSL, OPENSSH, GNUTLS, GNUPG, PHP, APACHE, IJG JPEG, LIBJPEG-TURBO and many more (cf. bug list on AFL's website [42]). Motivated by the success of fuzzing, we aim to use this technology for finding side-channel vulnerabilities. Typically, fuzzers use heuristic algorithms to mutate user-provided inputs to increase coverage, with the goal of finding crashes and other vulnerabilities. In contrast, DIFFUZZ uses fuzzing to perform a relational analysis, where the goal is to maximize the difference in resource usage for two copies of the program.

To realize this goal, an off-the-shelf fuzzer can be extended as follows: (1) the instrumentation is modified to collect additional information related to a resource consumption, such as timing, memory usage and response size; and (2) the *difference* between the costs observed for two program copies is recorded and sent back to the fuzzer, whose logic is modified to consider as *important* the inputs that increase this difference. In particular, the fuzzer maintains the so far observed difference *high-score* and prioritizes inputs leading to new high-score in addition to improved coverage, attempting to maximize the difference and thus find side-channels.

The timing cost is approximated by counting every (byte-code) instruction executed by the program. A similar cost is used in previous static analysis tools (THEMIS and BLAZER) allowing us to compare with them. Note that we can also measure the wall-clock time directly, by recording the execution time for each execution. The measurements can be performed on a clean, un-instrumented version of the program, using the inputs provided by fuzzing. However, we found that these measurements could sometimes be imprecise due to garbage collection and other processes running on the same machine. One can perform multiple runs for the same input, and take the average of these measurements, but we did not explore this direction further in this work, as we found that counting the instructions provides a good approximation.

Memory usage is measured by intermittent polling using a timer, which results in measuring the maximum consumption at any point during program execution. DIFFUZZ also measures response size (in bytes) for the values that are returned and the messages that are sent by the application.

We note that AFL supports programs written in C, C++, or Objective C. To make it applicable to JAVA programs, we use KELINCI [24], which provides an AFL-style instrumentation for JAVA programs, executes the instrumented programs and sends results back to a simple C program that interfaces with AFL. AFL does not know about the JAVA program in the background because it only communicates with the mentioned interface program, and hence, AFL can use its heuristics to generate inputs that are then executed on the Java programs.

E. Example

We illustrate the side-channel analysis on a password checking example. Listing 1 and Listing 2 show the code for the comparison of a user password with a server-side stored password in an *unsafe* and *safe* way respectively.

```
0  boolean pwcheck_unsafe(byte[] pub, byte[] sec) {
1    if (pub.length != sec.length) {
2      return false;
3    }
4    for (int i = 0; i < pub.length; i++) {
5      if (pub[i] != sec[i]) {
6        return false;
7      }
8    }
9    return true;
10 }
```

Listing 1: Unsafe Password Checking

```
0  boolean pwcheck_safe(byte[] pub, byte[] sec) {
1    boolean unused;
2    boolean matches = true;
3    for (int i = 0; i < pub.length; i++) {
4      if (i < sec.length) {
5        if (pub[i] != sec[i]) {
6          matches = false;
7        } else {
8          unused = true;
9        }
10     } else {
11       unused = false;
12       unused = true;
13     }
14   }
15   return matches;
16 }
```

Listing 2: Safe Password Checking

The *unsafe* variant contains a timing side channel because its early-return in lines 2 and 6. These two locations were fixed in the *safe* variant by iterating over the complete password, even when the two passwords already do not match at an earlier point. To apply DIFFUZZ, we built a driver for the *unsafe* variant based on Procedure 1 with length limit of 16 bytes for each fuzzed value (see Listing 3). The way we parse the input in this example ensures that all three values have same length. The field `Mem.instrCost` holds the current cost measured by the instrumentation, i.e., in our case the number of executed bytecode instructions. The method `Mem.clear()` resets the current cost, which is necessary to measure the cost for each execution separately. `Kelinci.addCost(diff)` tells the fuzzer to use the cost difference `diff` as cost metric during the input assessment. We did run the fuzzer for 30 minutes and obtained a maximum cost difference of 47 bytecode instructions, with the inputs shown in Listing 4.

The value of sec_2 is matching the complete value of pub, whereas the value of sec_1 is not matching at all. Note that the fuzzer generated these values on its own, without any further influence by the driver. The initial input file (the seed file), generated randomly, leads to the cost difference 0. In fact the difference of 47 instructions is the worst-case scenario and was already retrieved by the fuzzer within 69 seconds. A value greater than 0 was retrieved by the fuzzer within 5 seconds.

```
void driver(String[] args) {
  int maxLen = 16;
  int maxData = 3 * maxLen;
  byte[] allBytes = readDataUpToMax(args[0],
      maxData);
  int len = allBytes.length / 3;
  byte[] pub = Arrays.copyOfRange(allBytes, 0, len);
  byte[] sec_1 =Arrays.copyOfRange(allBytes, len,
      2*len);
  byte[] sec_2 = Arrays.copyOfRange(allBytes, 2*len
      , 3*len);

  Mem.clear();
  boolean answer1 = pwcheck_unsafe(pub, sec_1);
  long cost1 = Mem.instrCost;

  Mem.clear();
  boolean answer2 = pwcheck_unsafe(pub, sec_2);
  long cost2 = Mem.instrCost;

  long diff = Math.abs(cost1 - cost2);
  Kelinci.addCost(diff);
}
```

Listing 3: Password Checking Driver

```
pub=[−48, −4, −48, 7, 17, 0, −24, −48, −48, 16, −48, −3,
    108, 72, 32, 0]
sec_1=[72, 77, −16, −66, −48, −48, −48, −48, −28, 0, 100, 0,
    0, 0, 0, −48]
sec_2=[−48, −4, −48, 7, 17, 0, −24, −48, −48, 16, −48, −3,
    108, 72, 32, 0]
```

Listing 4: Input for Max Cost Difference after 30 min.

Afterwards, we used a similar fuzzing driver on the *safe* variant for 30 minutes as well, and we ran DIFFUZZ again. In this case we have observed no cost differences (i.e., $\delta = 0$). To further check that the program was indeed repaired, we executed the *safe* variant with the inputs obtained with the previous fuzzer run on the *unsafe* variant, obtaining again zero difference.

III. EVALUATION

To assess the effectiveness of DIFFUZZ in identifying side-channel vulnerabilities, we evaluated it on two sets of benchmarks. The first set, taken from [8] and [14], contains programs with known time and space side-channels, as well as repaired versions. The second set contains new complex examples from the DARPA Space/Time Analysis for Cybersecurity (STAC) program [20] as well as popular real-world applications, on which we identified *new* vulnerabilities.

For the first set of benchmarks, we compare DIFFUZZ with BLAZER [8] and THEMIS [14], two state-of-the-art static analysis tools for detecting side-channel vulnerabilities in JAVA programs. BLAZER uses decomposition techniques for proving bounded non-interference while THEMIS uses Quantitative Cartesian Hoare Logic reasoning to check bounded non-interference for JAVA programs, where the bound ϵ is set to either 0 or 64 [14]. Since BLAZER and THEMIS are not available, we perform the comparison on the same set of benchmarks that were used to evaluate the respective tools [8], [14]. We received the code for all the benchmarks from the THEMIS developers. We note that three examples were missing (Apache Shiro, Apache Crypto and bc-java); we therefore could not analyze them. Our tool and the benchmarks are available at our GitHub repository: https://github.com/isstac/diffuzz

A. Experimental Setup

For each target application, we wrote a driver in JAVA, following the steps in Procedure 1 (Section II). Note that by default, the instrumentor ignores all library code and hence, we specifically copied methods from libraries into the application to instrument them as well.

Due to the randomness in fuzzing, we run DIFFUZZ on each application five times, and we report the averaged results. All the experiments were performed on a server with OPENSUSE LEAP 42.3 featuring 8 Quad-Core-AMD 8384 2.7 GHz and 64 GB of memory. We used OPENJDK 1.8.0_151 and GCC 4.8.5. Although typically DIFFUZZ is able to identify a side-channel vulnerability in a few seconds, we run each experiment for 30 minutes.

As mentioned, DIFFUZZ reads the inputs to a program from an initial seed file. In general, we used a randomly generated file. Some applications get specific types of inputs, such as IBASys that needs an image file. In that case, we extracted the byte encoding from a random image and used it as the initial input file. For finding timing side-channels, we use a simple cost model that counts the bytecode instructions during the program run. Both BLAZER and THEMIS similarly count the instructions for their timing side-channel analysis.

B. Evaluating DIFFUZZ on the BLAZER Examples

We employed DIFFUZZ on the examples from [8] for evaluating BLAZER, which were also analyzed with THEMIS [14]. They consist of programs with timing side-channels and repaired *safe* versions. They are small applications with up to a hundred lines of code.

Note that the safe version of *unixlogin* was not executable due to a `NullPointerException` during hash comparison (cf. the Figure 3 in the BLAZER paper [8], second example line 7). Although THEMIS did not include this subject, we fixed the issue by adding a dummy comparison of the same MD5 hash of the provided password.

Results: We summarize the results in Table I. The *Average* column shows the (average) cost difference δ between two executions of an application. The *Time* column includes the time that each of the tools (DIFFUZZ, BLAZER, THEMIS) needed to identify a vulnerability. The numbers for BLAZER and THEMIS are extracted from [14]; for the THEMIS experiments, the bound ϵ was set to zero. For DIFFUZZ, the time shows the average earliest time that cost difference is bigger than zero, $\delta > 0$. The time values for some safe versions are not provided because in those cases the δ is zero.

The results indicate that DIFFUZZ is able to accurately identify all the side-channel vulnerabilities in the *unsafe* versions. The average cost difference for all *unsafe* programs is more than zero and sometimes it is very large.

DIFFUZZ behaves as expected on the majority of the *safe* versions, finding zero difference, but it also found some discrepancies. In two cases (*Array* and *unixlogin*) the differences found (1 and 3) may be attributed to slight discrepancies between the intermediate representations of the different analyses, and can thus be considered negligible. However, in two other cases DIFFUZZ found large δ values indicating that the repaired versions are in fact *not* safe. We discuss them below.

LoopAndbranch: Both BLAZER and THEMIS deemed the repaired version of *LoopAndbranch* function as *safe*.

DIFFUZZ instead identified a huge difference $\delta = 1,389,926,404$ in computed costs, which occurs due to integer overflow. In particular, the value assigned by the fuzzer to one of the secrets is the maximum integer value in Java, which gets added to 10, becoming a negative value. As a result, none of the loops in the code get executed and the cost is very small compared to the cost of the other execution, with the second secret value. This vulnerability, which was confirmed by the developers of BLAZER, highlights the importance of handling overflow in analysis tools.

gpt14: This function computes the modular exponentiation, $a^b mod(p)$, used for the encryption and decryption of messages. Here, a and p are public values and b is the secret.

BLAZER reported this example as safe for a non-zero bound whereas THEMIS reported it as safe for a zero bound (non-interference). DIFFUZZ found that even though the repair has substantially reduced the cost difference, still $\delta = 517$ (which is consistent with the BLAZER results). This vulnerability is due to an extra *if* statement that depends on the secret and it was confirmed by the Themis' developers.

C. Evaluating DIFFUZZ on the THEMIS Examples

We further evaluated DIFFUZZ on the larger JAVA programs with time and space side-channels from [14]. These programs have up to 20K LOC (although only some smaller parts were analyzed with all three tools), and are extracted from complex-real world applications, such as Tomcat, Spring-Security and Eclipse Jetty HTTP web server.

All benchmarks except *DynaTable*, *Advanced_table*, *OpenMRS* and *OACC* come with a repaired version. Some of the benchmarks (*Tomcat*, *pac4j*) include interactions with a database. In our experiments, we created the required databases and run them instead of simulating them with other data structures. We used the H2 database engine [3] to create an SQL database accessible via the JDBC API.

TABLE I: The results of applying DIFFUZZ to the BLAZER examples. Discrepancies are highlighted in red and italics.

Benchmark	Version	Average δ	Std. Error	Maximum	Time (s)		
					DIFFUZZ, $\delta > 0$	BLAZER	THEMIS
MicroBench							
Array	Safe	1.00	0.00	1	7.40 (+/- 1.21)	1.60	0.28
Array	Unsafe	192.00	2.68	195	7.40 (+/- 0.93)	0.16	0.23
LoopAndbranch	*Safe*	*1,468,212,312.40*	*719,375,479.77*	*4,278,268,702*	*18.60 (+/- 6.40)*	*0.23*	*0.33*
LoopAndbranch	Unsafe	4,283,404,852.40	4,450,278.15	4,294,838,782	10.60 (+/- 2.62)	0.65	0.16
Sanity	Safe	0.00	0.00	0	-	0.63	0.41
Sanity	Unsafe	4,213,237,198.00	60,857,888.00	4,290,510,883	163 (+/- 40.63)	0.30	0.17
Straightline	Safe	0.00	0.00	0.00	-	0.21	0.49
Straightline	Unsafe	8.00	0.00	8	14.60 (+/- 6.53)	22.20	5.30
unixlogin	Safe	3.00	0.00	3	510 (+/- 91.18)	0.86	-
unixlogin	Unsafe	2,880,000,008.00	286,216,701.00	3,200,000,008	464.20 (+/- 64.61)	0.77	-
STAC							
modPow1	Safe	0.00	0.00	0	-	1.47	0.61
modPow1	Unsafe	2,576.00	168.21	3,068	4.80 (+/- 1.11)	218.54	14.16
modPow2	Safe	0.00	0.00	9	-	1.62	0.75
modPow2	Unsafe	1,471.00	891.00	5,206	23 (+/- 3.48)	7813.68	141.36
passwordEq	Safe	0.00	0.00	0.00	-	2.70	1.10
passwordEq	Unsafe	86.40	20.31	127	8.60 (+/-2.11)	1.30	0.39
Literature							
k96	Safe	0.00	0.00	0	-	0.70	0.61
k96	Unsafe	338.00	185.13	3,087,339	3.40 (+/- 0.98)	1.29	0.54
gpt14	*Safe*	*163.20*	*79.84*	*517*	*4.20 (+/- 0.80)*	*1.43*	*0.46*
gpt14	Unsafe	6,673,760.00	2,211,811.00	12,965,890	4.40 (+/- 1.03)	219.30	1.25
login	Safe	0.00	0.00	0	-	1.77	0.54
login	Unsafe	62.00	0.00	62	10 (+/- 2.92)	1.79	0.70

Results: Table II displays our results; the results for THEMIS are taken from [14]. Once again, DIFFUZZ successfully identified vulnerabilities in the unsafe versions of these examples and for the majority of the repaired versions, DIFFUZZ found only small differences, as expected. In one case (jetty), DIFFUZZ identified a *new* vulnerability in the repaired version. Some other examples (Tomcat, pac4j, OACC) also show some discrepancies. We provide more details below.

Jetty: THEMIS was used to analyze a known vulnerability in the Eclipse Jetty HTTP web server and a repaired version of it. Furthermore, THEMIS found a similar vulnerability in another part of the Jetty application.

The original unsafe version of the code performs some checking over sensitive credential information by calling the built-in equality method provided by the `java.lang.String library`. Since this method returns false as soon as it finds a mismatch between two characters, it introduces a timing side-channel vulnerability. The method has been repaired with the one in Listing 5. This repair is very common and has been used in many implementations to avoid time channels.

```
0 boolean stringEquals(String s1, String s2) {
1   boolean result = true;
2   int l1 = s1.length();
3   int l2 = s2.length();
4   if (l1 != l2) result = false;
5   int n = (l1 < l2) ? l1 : l2;
6   for (int i = 0; i < n; i++)
7     result &= s1.charAt(i) == s2.charAt(i);
8   return result;
9 }
```

Listing 5: Jetty safe string comparison analyzed in [14]

Interestingly, for this example, DIFFUZZ found that it is still vulnerable with $\delta = 5,454$. The reason for this vulnerability is subtle. It turns out that the operation at line 7 is not constant time: it takes either 2 or 3 bytecodes, depending on the outcome of the equality check between the two characters (the operation is optimized for the case that the outcome is false). Although there is a difference of only one bytecode instruction, having this operation in the loop amplifies its impact. This could not be discovered by THEMIS because in its intermediate representation (Jimple), the operation at line 7 takes constant time. We further note that we imposed no constraints on the input and this is in line with the THEMIS experiments. The observed difference is proportional with the size of the input, and for small input sizes, both versions could be considered safe. However, for large input sizes, both safe and unsafe versions are in fact not safe.

Tomcat, pac4j, OACC: The vulnerability of *pac4j* is due to the encoding of a password, which is performed during user authentication, and is assumed to be expensive. Nevertheless, the code provided by the THEMIS developers did not include an expensive implementation of the password encoding (they instead used a model which was not provided to us). Since we did not use any models we could not find a noteworthy cost difference between the provided safe and unsafe versions (cf. Table II subject *pac4j Safe* and *pac4j Unsafe*). We also used another more expensive password encoding method, denoted with a star (*) in Table II, which iterates over the password, to get a stronger indication that there is an actual timing side-channel vulnerability (cf. Table II subject *pac4j Unsafe**).

We also found vulnerabilities in the unsafe versions of Tomcat and OACC, however the generated δs were small. Upon consulting with the THEMIS developers, it appears that,

TABLE II: Comparison against THEMIS

Benchmark	Version	DIFFUZZ				THEMIS		
		Average δ	Std. Error	Maximum	Time (s) $\delta > 0$	$\epsilon = 64$	$\epsilon = 0$	Time (s)
Spring-Security	Safe	1.00	0.00	1	9.00 (+/- 1.26)	✓	✓	1.70
Spring-Security	Unsafe	149.00	0.00	149	8.80 (+/- 1.16)	✓	✓	1.09
JDK7-MsgDigest	Safe	1.00	0.00	1	15.80 (+/- 3.93)	✓	✓	1.27
JDK6-MsgDigest	Unsafe	10,215.00	6,120.00	34,479	7.40 (+/- 1.29)	✓	✓	1.33
Picketbox	Safe	1.00	0.00	1	29.20 (+/-5.00)	✓	✗	1.79
Picketbox	Unsafe	4,954.00	1,295	8,794	16.80 (+/- 2.58)	✓	✓	1.55
Tomcat	Safe	12.20	1.61	14	13.80 (+/- 1.29)	✓	✗	9.93
Tomcat	*Unsafe*	*33.20*	*3.40*	*37*	*128.60 (+/- 87.20)*	✓	✓	*8.64*
Jetty	*Safe*	*5454.00*	*1330.88*	*8898*	*9.40 (+/- 1.86)*	✓	✓	*2.50*
Jetty	Unsafe	10786.60	2807.51	16020	7.00 (+/- 1.05)	✓	✓	2.07
orientdb	Safe	6.00	0.00	6	3.20 (+/- 0.97)	✓	✗	37.99
orientdb	Unsafe	6,604.00	3,681	19,300	3.00 (+/- 0.84)	✓	✓	38.09
pac4j	Safe	10.00	0.00	10	5.00 (+/- 1.22)	✓	✗	3.97
pac4j	*Unsafe*	*11.00*	*0.00*	*11*	*8.00 (+/- 2.76)*	✓	✓	*1.85*
pac4j	*Unsafe**	*39.00*	*0.00*	*39*	*10.80 (+/- 5.80)*	-	-	-
boot-auth	Safe	5.00	0.00	5	5.20 (+/- 0.20)	✓	✗	9.12
boot-auth	Unsafe	101.00	0.00	101	5.20 (+/- 0.20)	✓	✓	8.31
tourPlanner	Safe	0.00	0.00	0	-	✓	✓	22.22
tourPlanner	Unsafe	522.40	18.60	576	19.20 (+/- 0.80)	✓	✓	22.01
DynaTable	Unsafe	95.80	0.44	97	3.60 (+/- 1.21)	✓	✓	1.165
Advanced_table	Unsafe	92.40	1.54	97	11.20 (+/- 1.62)	✓	✓	2.01
OpenMRS	Unsafe	206.00	0.00	206	11.60 (+/- 3.22)	✓	✓	9.71
OACC	*Unsafe*	*47.00*	*0.00*	*47*	*7.00 (+/- 1.30)*	✓	✓	*1.83*

similar to *pac4j*, some manually built models were used, which we could not obtain.

D. Employing DIFFUZZ *on New Examples*

We also applied DIFFUZZ on new JAVA examples, including two complex applications taken from the Cybersecurity (STAC) program [20]: IBASys and CRIME, and two real-world open-source projects: Apache FtpServer [1] and AuthMeReloaded [2].

Results: Table III shows the results. For the reported zero-day vulnerabilities, all are confirmed and, collaborating with the developers and the community, solutions have been proposed and at this point most of them have been fixed. We explain our findings in more details.

CRIME: CRIME is an instance of the CRIME attack ("Compression Ratio Info-leak Made Easy") [19], which is as follows. Suppose a user is tricked into visiting a website `attack.com`, which has a malicious script making several requests to `bank.com`. Each request is a concatenation of public input generated by the script and the login (secret) cookie of the user. To avoid latency, protocols such as HTTPS and SPDY compress the requests before they are sent. The communication channel is encrypted, but the adversary can observe the size of the compressed package. When the public input is close to the secret, the compression is more efficient due to the redundancies, and the reduction in the size of the compressed package is more significant. Hence, the adversary can infer information about the secret. We analyzed the string compression procedure (160 LOC). It uses various input-output streams and involves complex string manipulations that are difficult to analyze with existing static analysis tools.

DIFFUZZ correctly identifies a space side-channel that reveals the secret through the size of the compressed output.

IBASys: IBASys is a network-based authentication server that uses images in place of textual passwords. To log in, a user supplies a username and a passcode image (*e.g.,* a JPEG image). Following a successful authentication, IBASys replies with a response containing an encrypted session token. This session token could then be used to interact with other services that rely on IBASys for their authentication needs. We analyzed the authentication procedure (707 LOC), which performs complex image manipulations.

DIFFUZZ managed to generate input files that are bytecode representations of valid images and it was also able to uncover a timing-channel that is due to early termination in a loop that matches the two (public and private) provided images. The maximum cost difference found by DIFFUZZ is $\delta = 262$, where the length of `image_public` is 18,995 bytes.

Apache FtpServer: We also applied DIFFUZZ on the open-source project Apache FtpServer [1], which has a very large code base; we focused our analysis on specific classes as reported below.

We identified a previously unknown timing side-channel in the class `ClearTextPasswordEncryptor` (115 LOC), in which the method `boolean matches(String, String)` uses the `String.equals` method for the comparison of the user provided password and the server-side stored password. This comparison returns false as soon as a character does not match, and hence, it could be used by a potential attacker to obtain knowledge about the hidden secret password. We have found this kind of vulnerability also in the classes `Md5PasswordEncryptor` (185 LOC) and `SaltedPasswordEncryptor` (211 LOC). We reported the issues to the developers who confirmed and fixed all of them. We also analyzed safe versions (provided by the developers) which fixed the issue. For all of them but one, the safe variant of string comparison did eliminate the vulnerability.

TABLE III: The results of applying DIFFUZZ on new examples

Benchmark	Version	Average δ	Std. Error	Maximum	Time (s) $\delta > 0$
STAC					
CRIME	unsafe	295.40	117.05	782	7.40 (+/- 1.12)
ibasys (imageMacher)	unsafe	191	20.88	262	6.20 (+/- 0.66)
Zero-day Vulnerabilities					
Apache ftpserver Clear	safe	1.00	0.00	1	7.20 (+/- 1.24)
Apache ftpserver Clear	unsafe	47.00	0.00	47	6.80 (+/- 1.07)
Apache ftpserver MD5	safe	1.00	0.00	1	4.20 (+/- 1.93)
Apache ftpserver MD5	unsafe	151.00	0.00	151	2.80 (+/- 1.11)
Apache ftpserver SaltedPW	(safe)	176.40	6.25	198	2.20 (+/- 0.73)
Apache ftpserver SaltedPW	unsafe	178.80	5.13	193	3.60 (+/- 1.08)
Apache ftpserver SaltedPW*	unsafe	163.40	3.80	178	5.40 (+/- 0.98)
Apache ftpserver StringUtils	safe	0.00	0.00	0	-
Apache ftpserver StringUtils	unsafe	53.00	0.00	53	3.00 (+/- 1.05)
AuthMeReloaded	safe	1.00	0.00	1	7.60 (+/- 0.75)
AuthMeReloaded	unsafe	383.00	0.00	383	9.20 (+/- 1.96)

For the class `SaltedPasswordEncryptor` DIFFUZZ still detected a vulnerability, so we continued our investigation and discovered that in addition to the matching method the used encryption method leaks information about the generated salt. We have thus analyzed method `String encrypt(String pw, String salt)`, marked with a star (*) in Table III. We are discussing with the developers with regard to this new vulnerability.

```
0 public final static String pad_unsafe(String src, char
      padChar, boolean rightPad, int totalLength) {
1     int srcLength = src.length();
2     if (srcLength >= totalLength) return src;
3     int padLength = totalLength − srcLength;
4     StringBuilder sb = new StringBuilder(
          padLength);
5     for (int i = 0; i < padLength; ++i) {
6         sb.append(padChar);
7     }
8     if (rightPad) {
9         return src + sb.toString();
10    } else {
11        return sb.toString() + src;
12    } }
```

Listing 6: Apache FtpServer StringUtils.pad unsafe version

Note that the salt in `SaltedPasswordEncryptor` gets randomly generated during encryption. Nevertheless for the `matches` method we fuzz the complete stored password including the salt. Furthermore, for the more focused analysis of the `encrypt` method we test if the algorithm leaks some information about the used salt via a side-channel.

We have also found a timing side-channel in the method `String StringUtils.pad(String, char, boolean, int)` (Listing 6), which was also confirmed by the developers. This method leaks the padding in a timing side-channel, from which a potential attacker could obtain the length of the `src` String. The padding is used to extend a username to fixed length, hence, a potential attacker could obtain the length of a given username, which might be used for further attacks. The vulnerability is caused by: (1) the early return in line 2, and (2) the for loop in line 5-7, which only runs for `padLength` iterations. The safe version, provided in our repository solves both issues.

AuthMeReloaded: We have also found an unknown timing side-channel in the open-source project AuthMeReloaded [2], which is an authentication plugin for Minecraft servers available on GitHub. It provides features like username spoof protection and anti-bot measures. Specifically, we found a vulnerability in the class `RoyalAuth` (209 LOC), in the inherited method `boolean comparePassword(String password, HashedPassword hashedPassword, String name)`. Similar vulnerabilities have been found in the classes `Sha256` and `Pbkdf2`. The developers fixed these vulnerabilities within a few days by using a constant time comparison algorithm.

E. Discussion

One advantage of DIFFUZZ compared to the other tools is that it not only shows whether an application is vulnerable, but also shows the magnitude of the vulnerability. This observation can be leveraged to estimate the severity of a vulnerability and it also makes it possible for the developers to compare different repaired versions of an application.

Analysis Time: Tables I, II and III also show the time that each of the tools needed for analysis. Both BLAZER and THEMIS were run on different hardware, making the timing reported incomparable. In general, static analysis is shown to be much faster than dynamic analysis; our results show that nonetheless DIFFUZZ is able to identify vulnerabilities in a reasonable time. In principle DIFFUZZ can run for a long time and it can still generate new inputs that increase the cost difference. However, we observed in preliminary test executions that our experiments find a plateau within 30 minutes, which is the time bound we applied.

For our experiments, we observed three different kinds of behavior: (a) DIFFUZZ identifies a small cost difference very fast, and it increases it over time; (b) DIFFUZZ identifies a big cost difference very fast and remains in a plateau after a few seconds; and (c) DIFFUZZ needs a long time to find a cost difference at all. Cases (a) and (b) were almost equally distributed on our experiments and covered almost all of them. Only for three experiments we observed case (c). As an illustration the plots in Figure 2 show the average maximum cost development within the first 5 minutes for the three cases.

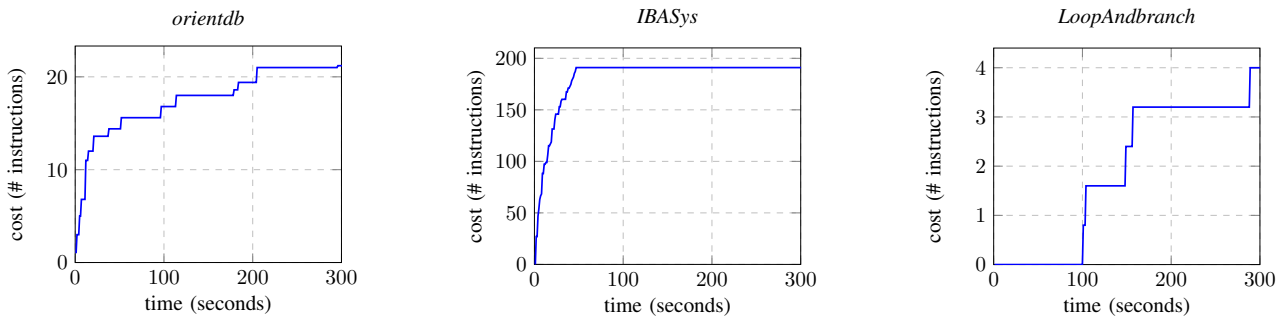

Fig. 2: Averaged cost over time for *orientdb*, *IBASys*, and *LoopAndbranch* (unsafe versions).

Orientdb (case (a)) checks passwords by comparing between user-given and stored passwords. The longer the matching prefix is, the higher will be the processing cost. Exact value matching is in general very difficult for fuzzing because it is hard to randomly generate the exact (unlikely) values that match the stored password. While DIFFUZZ finds quickly a small prefix, which reveals a cost difference greater than zero, it needs some time to reach a higher value.

For *IBASys* (case (b)), DIFFUZZ finds the maximum average value already after a few seconds, and thus leads very fast to the shown plateau value. The reason could be that the initial seed file guides the fuzzer already into a costly path or that the costly paths have a high probability, and hence, the fuzzer can easily catch them.

For *LoopAndbranch* (case (c)) DIFFUZZ reaches some parts of the code only with specific values for the secret and this is difficult to achieve with fuzzing. We believe that the limitations illustrated with cases (a) and (c) can be mitigated by adding further guidance to the fuzzer and by, e.g., combining fuzzing and symbolic execution.

Vulnerability vs Exploit: DIFFUZZ can identify side-channel vulnerabilities but can not assess whether they are exploitable by a real attack. The synthesis of a real attack, which would be necessary to assess the severity of the found vulnerability, is out of scope for this work. Nevertheless, we believe that our contribution is a first step in this direction.

IV. Related Work

DIFFUZZ is related to a large body of work on checking non-interference via self-composition [10]. For instance, related work [7] presents a self-composition approach to timing-channel analysis, which however does not check bounded non-interference. We already compared with the most recent related tools, BLAZER [8] and THEMIS [14].

CoCoChannel [12] uses static analysis for finding side-channel vulnerabilities and presents a comparison with THEMIS and BLAZER on the same benchmarks, showing better scalability. While CoCoChannel also found discrepancies in the THEMIS and BLAZER benchmarks, the approach still fails to report vulnerabilities for the repaired versions in, e.g., loopAndBranch and jetty.

Stacco [40] also uses a differential analysis for finding timing side-channels, using random inputs. However, Stacco does not perform directed fuzzing, it does not check bounded non-interference, and it does not address Java.

There is a large amount of related work on side-channel analysis, for example [6], [13], [15], [17], [25], [26], [33]. The most successful approaches use abstract interpretation (for cache side-channels analysis) [18], [27], [32] and are thus quite different than DIFFUZZ. Other techniques [9], [35], [37] use symbolic execution and constraint solving with model counting for quantifying side-channel leakage and for synthesis of attacks. They address JAVA programs, but may have scalability issues, due to the expensive constraint manipulation.

Other related techniques aim to quantify leakage using Monte Carlo sampling [16], [23]. In contrast to DIFFUZZ, these techniques provide *quantitative* results, but they may be imprecise in practice.

Fuzzing has received renewed interest in the software engineering community, with many recent approaches reported [29], [30], [34], [39]. Most related are techniques that use fuzzing alone [29], [39] or a combination of fuzzing and symbolic execution [34] to analyze the algorithmic complexity of programs, by monitoring a resource consumption. In particular, Badger [34] also uses Kelinci and AFL for the fuzzing part. None of these works address side-channel analysis.

V. Conclusions and Future Work

We presented DIFFUZZ, the first differential fuzzing approach for automatically finding side-channel vulnerabilities. We have shown that DIFFUZZ can keep up with existing approaches such as BLAZER and THEMIS. Furthermore, DIFFUZZ found new vulnerabilities in popular open-source JAVA applications such as Apache FtpServer. In the future, we plan to explore automated repair methods to eliminate the vulnerabilities discovered with DIFFUZZ. Additionally, we plan to augment our work with statistical guarantees similar to the STADS framework [11].

Acknowledgment

This material is based on research sponsored by DARPA under agreement number FA8750-15-2-0087. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. This work is also supported by the German Research Foundation (GR 3634/4-1 EMPRESS).

REFERENCES

[1] Apache FtpServer. https://mina.apache.org/ftpserver-project/. Accessed: 2018-08-21.

[2] Authentication plugin for the Bukkit/Spigot API. https://github.com/AuthMe/AuthMeReloaded. Accessed: 2018-08-21.

[3] H2 database engine. http://www.h2database.com/html/main.html. Accessed: 2018-05-06.

[4] The Meltdown Attack. https://meltdownattack.com/. Accessed: 2018-08-21.

[5] Xbox 360 Timing Attack. http://beta.ivc.no/wiki/index.php/Xbox_360_Timing_Attack. Accessed: 2018-08-21.

[6] Dakshi Agrawal, Josyula R. Rao, and Pankaj Rohatgi. Multi-channel Attacks. In Colin D. Walter, Çetin Kaya Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2003, 5th International Workshop, Cologne, Germany, September 8-10, 2003, Proceedings*, volume 2779 of *Lecture Notes in Computer Science*, pages 2–16. Springer, 2003.

[7] B. Almeida, M. Barbosa, J. S. Pinto, and B. Vieira. Formal verification of side-channel countermeasures using self-composition. In *Science of Computer Programminga78(7)*, 2013.

[8] Timos Antonopoulos, Paul Gazzillo, Michael Hicks, Eric Koskinen, Tachio Terauchi, and Shiyi Wei. Decomposition instead of self-composition for proving the absence of timing channels. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 362–375, 2017.

[9] Lucas Bang, Abdulbaki Aydin, Quoc-Sang Phan, Corina S. Păsăreanu, and Tevfik Bultan. String Analysis for Side Channels with Segmented Oracles. In *Proc. of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 193–204, New York, NY, USA, November 2016. ACM.

[10] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. In *Proceedings of the 17th IEEE Workshop on Computer Security Foundations*, CSFW '04, pages 100–, Washington, DC, USA, 2004. IEEE Computer Society.

[11] Marcel Böhme. STADS: Software testing as species discovery. *ACM Transactions on Software Engineering and Methodology*, 27(2):7:1–7:52, June 2018.

[12] Tegan Brennan, Seemanta Saha, Tevfik Bultan, and Corina S. Păsăreanu. Symbolic path cost analysis for side-channel detection. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, pages 27–37, New York, NY, USA, 2018. ACM.

[13] David Brumley and Dan Boneh. Remote Timing Attacks Are Practical. In *Proc. of the 12th Conf. on USENIX Security Symposium - Volume 12*, SSYM'03, Berkeley, CA, USA, 2003. USENIX Association.

[14] Jia Chen, Yu Feng, and Isil Dillig. 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 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 875–890, 2017.

[15] Shuo Chen, Rui Wang, XiaoFeng Wang, and Kehuan Zhang. Side-Channel Leaks in Web Applications: A Reality Today, a Challenge Tomorrow. In *Proc. of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 191–206, Washington, DC, USA, 2010. IEEE Computer Society.

[16] Tom Chothia, Yusuke Kawamoto, and Chris Novakovic. LeakWatch: Estimating Information Leakage from Java Programs. In *19th European Symposium on Research in Computer Security - Volume 8713*, ESORICS 2014, pages 219–236, New York, NY, USA, 2014. Springer-Verlag New York, Inc.

[17] Quoc Huy Do, Richard Bubel, and Reiner Hähnle. Exploit Generation for Information Flow Leaks in Object-Oriented Programs. In *ICT Systems Security and Privacy Protection: 30th IFIP TC 11 Intl. Conf., SEC 2015, Hamburg, Germany*, pages 401–415. Springer, 2015.

[18] Goran Doychev, Dominik Feld, Boris Köpf, Laurent Mauborgne, and Jan Reineke. CacheAudit: A Tool for the Static Analysis of Cache Side Channels. In *Proc. of 22nd USENIX Conf. on Security*, SEC'13, pages 431–446, Berkeley, CA, USA, 2013. USENIX Association.

[19] Thai Duong and Juliano Rizzo. The CRIME attack. In *Presentation at ekoparty Security Conf.*, 2012.

[20] Mr. Dustin Fraze. Space/Time Analysis for Cybersecurity (STAC). https://www.darpa.mil/program/space-time-analysis-for-cybersecurity. Accessed: 2018-08-21.

[21] Matthias Höschele and Andreas Zeller. Mining input grammars from dynamic taints. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 720–725, New York, NY, USA, 2016. ACM.

[22] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space aslr. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 191–205. IEEE, 2013.

[23] Yusuke Kawamoto, Fabrizio Biondi, and Axel Legay. Hybrid Statistical Estimation of Mutual Information for Quantifying Information Flow. In *FM*, volume 9995 of *Lecture Notes in Computer Science*, pages 406–425, 2016.

[24] Rody Kersten, Kasper Luckow, and Corina S. Păsăreanu. Poster: Afl-based fuzzing for java with kelinci. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 2511–2513, New York, NY, USA, 2017. ACM.

[25] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Proc. of the 16th Annual International Cryptology Conf. on Advances in Cryptology*, CRYPTO '96, pages 104–113, London, UK, UK, 1996. Springer-Verlag.

[26] Boris Köpf and David Basin. An Information-theoretic Model for Adaptive Side-channel Attacks. In *Proc. of the 14th ACM Conf. on Computer and Communications Security*, CCS '07, pages 286–296, New York, NY, USA, 2007. ACM.

[27] Boris Köpf, Laurent Mauborgne, and Martín Ochoa. Automatic quantification of cache side-channels. In *Proc. of the 24th international Conf. on Computer Aided Verification*, CAV'12, pages 564–580, Berlin, Heidelberg, 2012. Springer-Verlag.

[28] Nate Lawson. Timing attack in Google Keyczar library. https://rdist.root.org/2009/05/28/timing-attack-in-google-keyczar-library/, 2009. Accessed: 2018-08-21.

[29] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. Perffuzz: Automatically generating pathological inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, pages 254–265, New York, NY, USA, 2018. ACM.

[30] Caroline Lemieux and Koushik Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, New York, NY, USA, 2018. ACM.

[31] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 605–622. IEEE, 2015.

[32] Heiko Mantel, Alexandra Weber, and Boris Köpf. A systematic study of cache side channels across aes implementations. In Eric Bodden, Mathias Payer, and Elias Athanasopoulos, editors, *Engineering Secure Software and Systems*, pages 213–230, Cham, 2017. Springer International Publishing.

[33] P. Mardziel, M. S. Alvim, M. Hicks, and M. R. Clarkson. Quantifying information flow for dynamic secrets. In *2014 IEEE Symposium on Security and Privacy (SP)*, pages 540–555, May 2014.

[34] Yannic Noller, Rody Kersten, and Corina S. Păsăreanu. Badger: Complexity analysis with fuzzing and symbolic execution. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, pages 322–332, New York, NY, USA, 2018. ACM.

[35] Corina S Pasareanu, Quoc-Sang Phan, and Pasquale Malacaria. Multi-run side-channel analysis using symbolic execution and max-smt. In *Computer Security Foundations Symposium (CSF), 2016 IEEE 29th*, pages 387–400. IEEE, 2016.

[36] Theofilos Petsios, Adrian Tang, Salvatore J. Stolfo, Angelos D. Keromytis, and Suman Jana. NEZHA: efficient domain-independent differential testing. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 615–632, 2017.

[37] Quoc-Sang Phan, Lucas Bang, Corina S. Pasareanu, Pasquale Malacaria, and Tevfik Bultan. Synthesis of adaptive side-channel attacks. In *30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, August 21-25, 2017*, pages 328–342, 2017.

[38] S. Sivakorn, G. Argyros, K. Pei, A. D. Keromytis, and S. Jana. Hvlearn: Automated black-box analysis of hostname verification in ssl/tls implementations. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 521–538, May 2017.

[39] Jiayi Wei, Jia Chen, Yu Feng, Kostas Ferles, and Isil Dillig. Singularity: Pattern fuzzing for worst case complexity. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, New York, NY, USA, 2018. ACM.

[40] Yuan Xiao, Mengyuan Li, Sanchuan Chen, and Yinqian Zhang. Stacco: Differentially analyzing side-channel traces for detecting ssl/tls vulnerabilities in secure enclaves. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 859–874, New York, NY, USA, 2017. ACM.

[41] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 283–294, New York, NY, USA, 2011. ACM.

[42] Michal Zalewski. American fuzzy lop (afl). http://lcamtuf.coredump.cx/afl/, 2014. Accessed: 2018-05-06.