1dFuzz: Reproduce 1-Day Vulnerabilities with Directed Differential Fuzzing

Songtao Yang
Tsinghua University
Beijing, China
yst18@mails.tsinghua.edu.cn

Yubo He
State Key Laboratory of Mathematical Engineering and Advanced Computing
Zhengzhou, China
heyubo9@hotmail.com

Kaixiang Chen
Tsinghua University
Beijing, China
ckx18@mails.tsinghua.edu.cn

Zheyu Ma
Tsinghua University
Beijing, China
mzy20@mails.tsinghua.edu.cn

Xiapu Luo
Hong Kong Polytechnic University
Hong Kong, China
csxluo@comp.polyu.edu.hk

Yong Xie
Qinghai University
Xining, China
mark.y.xie@qq.com

Jianjun Chen
Tsinghua University, Zhongguancun Laboratory
Beijing, China
jianjun@tsinghua.edu.cn

Chao Zhang
Tsinghua University, BNRist, Zhongguancun Laboratory
Beijing, China
chaoz@tsinghua.edu.cn

ABSTRACT

1-day vulnerabilities are common in practice and have posed severe threats to end users, as adversaries could learn from released patches to find them and exploit them. Reproducing 1-day vulnerabilities is also crucial for defenders, e.g., to block attack traffic against 1-day vulnerabilities. A core question that affects the effectiveness of recognizing and triggering 1-day vulnerabilities is *what is the unique feature of a security patch*. After conducting a large-scale empirical study, we point out that a common and unique feature of patches is the *trailing call sequence (TCS)* and present a novel directed differential fuzzing solution `1dFuzz` to efficiently reproduce 1-day vulnerabilities in this paper. Based on the TCS feature, we present a locator `1dLoc` able to find candidate patch locations via static analysis, a novel TCS-based distance metric for directed fuzzing, and a novel sanitizer `1dSan` able to catch PoCs for 1-day vulnerabilities during fuzzing. We have systematically evaluated `1dFuzz` on a set of real-world software vulnerabilities in 11 different settings. Results show that `1dFuzz` significantly outperforms state-of-the-art (SOTA) baselines and could find up to 2.26x more 1-day vulnerabilities with a 43% shorter time.

CCS CONCEPTS

• **Security and privacy → Systems security**.

KEYWORDS

1-day vulnerability, patch, differential testing, directed fuzzing

ISSTA '23, July 17–21, 2023, Seattle, WA, USA

ACM ISBN 979-8-4007-0221-1/23/07.
https://doi.org/10.1145/3597926.3598102

ACM Reference Format:
Songtao Yang, Yubo He, Kaixiang Chen, Zheyu Ma, Xiapu Luo, Yong Xie, Jianjun Chen, and Chao Zhang. 2023. `1dFuzz`: Reproduce 1-Day Vulnerabilities with Directed Differential Fuzzing. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597926.3598102

1 INTRODUCTION

Software inevitably has vulnerabilities, which are continuously found and fixed over time. Existing research [7, 38, 51] showed that most users do not upgrade and patch programs in time and are vulnerable to the known and patched vulnerabilities, i.e., *1-day vulnerabilities*, and the duration of this vulnerable time window is very long (e.g., several months or even years), making it a huge attack surface in practice. Considering the software supply chain scenario, this 1-day vulnerabilities issue gets much worse. Commonly, an application has external library dependencies (either with source code or binary), which in general will not get updated in time and often have 1-day vulnerabilities, which may hide for several years and endanger many users [33, 41, 42]. On the other hand, 1-day vulnerabilities are easier to discover and exploit than 0-day vulnerabilities, since adversaries could study released patches to identify 1-day vulnerabilities and generate exploits accordingly. Thus, 1-day vulnerability exploitation is a real and severe threat in practice.

To mitigate such threats, a straightforward and effective solution is applying patches, but it relies on the security awareness of users and is not reliable. A more proactive solution is scanning 1-day vulnerabilities in the software supply chain from the system perspective or deploying network-based intrusion detection systems to block attack traffic against the vulnerabilities from the network perspective. However, such solutions require comprehensive knowledge of the vulnerabilities, especially the proof-of-concept (PoC) inputs that can trigger the vulnerabilities. Thus, defenders must first reproduce PoCs for the patched 1-day vulnerabilities without source code, which have to answer the following two questions:

First, PoCs for 1-day vulnerabilities have to reach the patch locations, but *where are the patches in the patched binary?* Existing *binary-based* patch recognition solutions rely on binary diffing to find differences between the unpatched and patched binaries to locate patches, including static diffing solutions [2, 22] that utilize control flow graph (CFG) comparison, dynamic diffing solutions [18, 32] that extract features from program execution traces, and AI-based solutions [15, 17] that rely on deep learning to extract features of binary code. However, existing solutions all have difficulty in precisely recognizing patches because (1) compilation optimizations could cause many differences in binaries, even for the same code without patches, and (2) other types of code updates (e.g., new functionalities) rather than bug fixing will also cause differences in binaries. Such differences are hard to get separated from security patches. This problem could be relieved if the unique feature of patches (rather than binary differences) could be recognized.

Second, how to generate PoCs to trigger patches? PoCs for 1-day vulnerabilities behave differently at the patch locations, i.e., trigger abnormal behaviors in the unpatched binary and get blocked by the patch in the patched binary. Given a candidate patch location, there are two common ways to generate PoCs: symbolic execution and directed fuzzing. Symbolic execution solutions (e.g., APEG [9] and [16]) collect and solve constraints on paths reachable to the patch to generate PoCs. However, they cannot scale to large real-world applications. Directed fuzzing solutions (e.g., SemFuzz [49] and 1dVul [34]) utilize metrics like code coverage and CFG distance to guide the fuzzer to get closer to the patch location. However, such solutions lack sanitizers to determine whether the vulnerabilities are triggered or not during fuzzing. It will help design proper sanitizers if we could recognize the unique feature of patches.

In summary, to reproduce 1-day vulnerabilities, a core question to answer is: *what is the unique feature of a security patch?* The answer to this question affects the effectiveness of both recognizing patch locations and generating PoCs to trigger vulnerabilities. In this paper, we present an answer to this question and propose a novel and practical directed differential fuzzing solution `1dFuzz` to efficiently reproduce 1-day vulnerabilities in binaries.

First, based on a large-scale study (in §3.3), we figured out that 71.3% of patches check the validity of inputs and exit soon if inputs are invalid. In other words, a PoC often yields different execution traces *after* the patch location in the patched and unpatched binaries. A simpler form is that, the last N functions in these two execution traces, denoted as two trailing call sequences (TCS), are different. We point out that *TCS is a common and unique feature of patches.*

With this unique feature, we present a novel solution `1dLoc` to locate candidate patch locations by analyzing the call graphs. Specifically, we build the call graphs of both the patched and unpatched binaries, align functions in the two binaries, and find aligned functions that have different callee function sequences (i.e., candidate trailing call sequences) via static analysis. These aligned functions will be marked as the candidate patched functions.

Further, we adopt directed fuzzing to generate PoCs for candidate patches recognized by `1dLoc`. To detect whether a test case successfully triggers the patch during fuzzing, we present a novel TCS-based sanitizer `1dSan`. Specifically, we apply differential testing on the patched and unpatched binaries, track two sequences of functions that are invoked during testing, and report a PoC if the last N functions in these two sequences differ. In addition to the sanitizer, we present a new distance metric based on the function call relationship to guide directed fuzzing and improve efficiency.

We have built and systematically evaluated a prototype of `1dFuzz`, and compared it with baselines in 11 different settings, i.e., different sanitizers, different patch locators, and different fuzzers, on 242 pairs of pre-patch and post-patch binaries. For each setting of fuzzer and each pair of binaries, we run the binaries on two CPU cores for 24 hours and repeat each experiment three times. In total, we spent over 383,328 CPU hours. The results showed that, compared to baseline fuzzers, `1dFuzz` could find 2.71x-3.26x more 1-day vulnerabilities, and the average time to generate PoCs is 35.2%-43% shorter. In terms of code coverage, `1dFuzz` also outperforms the baselines in both speed and coverage. It also shows that our patch locator and runtime sanitizer all outperform existing solutions, respectively, and can be applied to promote existing fuzzers.

In summary, this paper makes the following contributions:

- We studied a large number of patches and pointed out that TCS is a unique and common feature of patches *for the first time.*
- We present a novel TCS-based sanitizer `1dSan` able to catch PoCs for 1-day vulnerabilities at runtime, a TCS-based locator `1dLoc` able to recognize candidate patch locations via static analysis, and a new distance metric for directed fuzzing.
- We presented a directed differential fuzzing solution `1dFuzz` for reproducing 1-day vulnerabilities in binaries.
- We thoroughly evaluated the performance of `1dFuzz` and showed that it significantly outperforms SOTA baselines.

2 BACKGROUND AND RELATED WORK

2.1 Locating Security Patches

The most common solution to locate security patches is binary diffing, which can be classified into the following three categories.

Static binary diffing. Differences between the patched and unpatched binaries are utilized to locate the patch. First, structural representations of programs (e.g., CFG) can be leveraged to locate binary differences. BinDiff [2] is the most widely used binary diffing tool. It matches functions in two binaries based on a series of function- and basic block-level structural similarity comparison algorithms. BinHunt [22] lifts binaries to the intermediate representation (IR) and calculates the similarity. Second, the semantics (e.g., symbolic constraints) of binaries could also be leveraged to perform binary diffing. CoP [30] analyzes dependencies between the input and output of basic blocks with symbolic execution and compares the similarity of code. Esh [14] considers the similarity of two procedures in stripped binaries by dividing them into strands of basic block slices. In general, static binary diffing solutions are limited by the accuracy and complexity of static analysis, not to mention that binary differences are not the unique feature of patches.

Dynamic binary diffing. Unlike static binary diffing solutions, dynamic binary diffing solutions compare binaries by extracting features from execution traces. Bayer et al. [8] propose scalable clustering that executes the binaries, adds records, and compares the behavioral profile of resources. Wang et al. [46] lift binaries to IR and instrument the IR with analysis code to collect traces. Blex [18] collects and compares the execution side effects of each function, including reads and writes to the stack and heap. BinSim [32] records

the system call sequences, aligns them based on critical system calls, and applies slicing to reduce noises of differences.

AI-based binary diffing. Deep learning has been thoroughly explored to assist binary diffing in recent years, showing better performance than conventional static and dynamic analysis solutions. IMF-SIM [45] dynamically collects the traces of all functions and trains a machine-learning model based on the labeled feature vectors of memory access offsets and system calls to infer the similarity of the two binaries. Gemini [48] constructs the attributed control flow graph and performs graph embedding to extract and compare features of binaries. VulSeeker-pro [23], Asm2Vec [15], InnerEye [54], αDiff [29], jTrans [44], and DeepBinDiff [17] all provide different variants of AI-based solutions to extract features of binaries and compare these features to locate the patch. However, existing AI-based solutions are only aware of differences at the function level and thus can not identify the instruction-level patch.

2.2 Generating PoCs for Patches

Given a candidate patch location, there are two types of solutions to generating PoCs for the patch.

Symbolic execution. APEG [9] investigates the inconsistency between unpatched and patched binaries and conducts symbolic execution to generate PoCs. Ma et al. [31] propose directed symbolic execution to generate input that reaches specified code lines. Dinges and Agha [16] propose combining symbolic backward execution and concrete forward execution to generate PoCs.

Directed fuzzing. SemFuzz [49] extracts semantic knowledge from the CVE descriptions of Linux kernel bugs and guides the fuzzing process with the knowledge. CAFL [26] uses changelogs to assist in directed greybox fuzzing. 1dVul [34] combines distance-guided directed fuzzing and dominator-guided directed symbolic execution, able to solve constraints to breakthrough bottlenecks of fuzzers. BEACON [25] prunes most executing paths and achieves an average 11.5x faster speed than previous directed fuzzers.

2.3 Fuzzing

Coverage-guided fuzzing. Greybox fuzzing tools [1, 11, 13, 21, 27] utilize coverage to guide the fuzzer to explore more code, but in general their ability to explore specific code is limited.

Directed fuzzing provides the capability of exploring specific code. AFLGo [10], the first of its kind, utilizes control-flow distance to guide the fuzzer to get closer to given targets. Hawkeye [12] improves AFLGo by introducing adjacent-function distances in power scheduling. ParmeSan [55] uses sanitizer instrumentation sites to improve the efficiency of exploration. CAFL [26] considers both target sites and data conditions to trigger restricted vulnerabilities.

Differential testing. Differential testing runs different programs in parallel and observes differences in the execution results to catch specific bugs. Nezha [43] records the path differences to guide fuzzing and find semantic bugs. EXPOZZER [35] employs diversified dual execution to report memory corruption bugs.

2.4 Sanitizers

Sanitizers are essential tools for detecting specific behaviors during execution. ASan [36] and its successors [5, 19] focus on detecting spatial and temporal memory safety violations. TSan [37] focuses on detecting data race bugs by recording the memory access and utilizing dynamic annotations to detect risky synchronizations.

```
1  --- a/libtiff/tif_getimage.c
2  +++ b/libtiff/tif_getimage.c
3  static int gtTileContig(TIFFRGBAImage* img,
4                          uint32 w, uint32 h, ...) {
5    ...
6    TIFFGetField(tif, TIFFTAG_TILEWIDTH, &tw);
7  @@ -645,6 +646,12 @@
8
9    flip = setorientation(img);
10   if (flip & FLIP_VERTICALLY) {
11 +     if ((tw + w) > INT_MAX) {
12 +       TIFFErrorExt(tif->tif_clientdata,
13 +                    TIFFFileName(tif), "%s",
14 +                    "tile size (too wide)");
15 +       return (0);
16 +     }
17     y = h - 1;
18     toskew = -(int32)(tw + w);
19   }
20   ...
21   _TIFFfree(buf);
22   return (ret);
23 }
```

Listing 1: Patch for the vulnerability CVE-2020-35523.

MSan [40] can detect uninitialized memory accesses by tracking the status of memory initialization through a shadow area. UBSan [6] instruments the program to capture undefined runtime behaviors.

3 TCS: TRAILING CALL SEQUENCE

In this section, we introduce the unique feature of patches and demonstrate how common it is via a large-scale empirical study.

3.1 Motivating Example

Take the vulnerability CVE-2020-35523 of `libtiff` (Listing 1) as an example. The vulnerability is an integer overflow at line 18, where `tw` and `w` are affected by input. The patch at line 11 verifies whether the sum of `tw` and `w` is larger than the maximum integer of word size. If yes, it calls the error handling function `TIFFErrorExt()` at line 12 and exits the current function at line 15, preventing subsequent vulnerable code execution. This is a typical workflow of patches, i.e., checking the validity of inputs, reporting errors, and quitting the normal execution flow.

3.2 The TCS Feature

The motivating example shows that the pre-patch (i.e., unpatched) and post-patch (i.e., patched) binaries will have different execution traces after the patch location if they are fed with the same PoC for the vulnerability. Even if we do not know the patch location, we can still observe that *the trailing part of function call sequences of the pre-patch and post-patch binaries are different.*

Definition 1. *Trailing Call Sequence (TCS). Given a program* `P` *and an input* `I`*, we denote the last* `N` *function calls before program* `P` *exits as the trailing call sequence* `TCS(P,I,N)`.

As shown in the motivating example, if a PoC for the patch is fed to the pre- and post-patch binaries, their TCSes are different. For simplicity, we denote this attribute as *the TCS feature of the patch.*

3.3 Popularity of the TCS Feature

Note that not all patches have the TCS feature. For instance, some patches simply correct the mistakes (e.g., setting all dangling pointers to NULL) or correct the wrong program states and then continue

Table 1: Code patterns used for TCS feature classification.

Code Pattern	Patch Pattern
if-exit	**IF** <condition> **THEN** `exit()`
if-return	**IF** <condition> **THEN** `return`
if-goto	**IF** <condition> **THEN** `GOTO` <label>
if-body	**IF** <condition> **THEN** <· · ·>

Table 2: Statistics of patches regarding to their code patterns and the TCS feature. Among 1031 patches in the benchmark, there are 735 (about 71.3%) patches having the TCS feature.

Code Pattern	Total Patches		Patches with TCS		Patches without TCS	
	#	ratio to total	#	ratio	#	ratio
if-exit	25	2.4%	25	100%	0	0%
if-return	437	42.4%	415	95.0%	22	5.0%
if-goto	147	14.3%	118	80.3%	29	19.7%
if-body	229	22.2%	151	65.9%	78	34.1%
Misc.	193	18.7%	26	13.5%	167	86.5%
Total	1031	100%	735	71.3%	296	28.7%

execution. On the other hand, not all code changes that have the TCS feature are patches. Thus, we conducted a large-scale empirical study on existing patches to learn the popularity of the TCS feature.

Benchmarks of patches. From an existing dataset [47] consisting of all CVE entries from 1999 to 2018 with patch reference links, we collect all vulnerable programs that are written in C/C++ and have memory safety-related vulnerabilities, and get 1031 patches belonging to 223 projects. We manually verified all 1031 patches to confirm their code patterns and whether they have the TCS feature.

Code patterns of patches. Table 1 shows four common code patterns of patches we found in the benchmark. The most common pattern of patches is `if-return`, as shown in the example in Listing 1. Such patches add a sanity check (or update the condition of an existing check) and terminate the current function early via `return` if the check fails. This type of patches are likely to have the TCS feature since such checks will change the trailing call sequence when the PoC satisfies the sanity check condition. Similarly, some patches have the `if-exit` and `if-goto` patterns, which terminate or alter the control flow via `exit` or `goto` if the updated sanity check fails. These two types of patches are also likely to have the TCS feature. Another pattern is denoted as `if-body`. Such patches also add or update sanity checks, but do not directly terminate the control flow if the sanity check fails. Some of them will call another exception handling function to handle the exception, which may also have the TCS feature. In addition, there is a small number (18.7%) of patches that do not add or update sanity checks. For instance, a patch may directly set dangling pointers to NULL at proper locations. Such patches in general do not have the TCS feature.

Distribution of patch patterns and TCS features. Table 2 shows the statistics data of the patches. Among the 1031 patches, only 193 (18.7%) patches do not update the sanity checks. It is consistent with the conclusion from the previous study PatchScope [52], i.e., most patches will add or modify input sanitizing checks. Among the rest 838 patches, after the sanity check fails, 25 of them will exit, 437 will return, 147 will goto somewhere, and 229 proceed in other ways (e.g., call exception handlers).

Among the 1031 patches, we found that 735 (71.3%) patches have the TCS feature. Specifically, all 25 patches of the `if-exit` pattern have the TCS feature. 415 (i.e., 95.0%) patches of the `if-return` pattern have the TCS feature, while 118 (i.e., 80.3%) patches of the `if-goto` pattern, 151 (i.e., 65.9%) patches of the `if-body` pattern, 26 (i.e., 13.5%) patches of the rest have the TCS feature.

In summary, patches of the `if-exit`, `if-return`, and `if-goto` patterns are highly likely to have the TCS feature. Even for patches of the `if-body` pattern, we also found that 65.9% of them have the TCS feature, because these patches may introduce new handler function calls in the trailing call sequence.

4 DESIGN

4.1 Overview

Based on the TCS feature, we present a novel directed differential testing solution `1dFuzz` to reproduce 1-day vulnerabilities. Figure 1 shows the overview of our solution.

Based on an unpatched (i.e., pre-patch) binary and a patched (i.e., post-patch) binary, `1dFuzz` first tries to align them, i.e., matching one function in a binary to its counterpart in another binary, in order to remove noises introduced by compilations and optimizations and enable further TCS-related analysis. Then, `1dFuzz` compares the call graphs of these binaries and identifies functions that have the TCS feature with static analysis via the patch recognition module `1dLoc`. Lastly, `1dFuzz` conducts a directed differential fuzzing to explore the candidate patch locations, catches PoCs for 1-day vulnerabilities with the TCS-based sanitizer `1dSan`, and tunes its strategy of directed fuzzing with a TCS-based distance metric. Details of these modules are listed as follows.

4.2 Binary Code Alignment

The patched and unpatched binaries could vary widely, even if there are only slight changes in the source code. This is because different construction settings, i.e., optimizations, are applied, which makes it hard to align code context with similarity-based solutions. Therefore, instead of existing heuristic-based or AI-based solutions, we present a new call graph-based binary code alignment solution.

4.2.1 Call Graph Construction. Building a complete and accurate call graph for a binary is an open challenge since the targets of indirect calls are hard to determine. `1dFuzz` mitigates this issue by constructing call graphs for binaries with a two-stage solution.

First, it disassembles the binary and recognizes all functions and direct call sites, then statically builds the sketch of the call graph based on the relationship of direct function calls. Second, it utilizes dynamic testing to collect indirect call relationships. For each indirect call in the execution trace, we add one edge from the caller function to the indirect callee function to the sketch call graph. The dynamic testing could also help us recognize more code and correct errors in our disassembly stage.

Note that `1dFuzz` will perform directed differential fuzzing, which is a kind of dynamic testing. Thus, we could reuse it to collect functions. In other words, `1dFuzz` will use the sketch call graph to assist further TCS analysis and fuzzing and then extract information from the fuzzing traces to complement the sketch call graph. This process runs iteratively, so the call graph grows gradually. However, we do not guarantee that the call graph built this way is complete.

4.2.2 Binary Function Alignment. Given the call graphs of both patched and unpatched programs, `1dFuzz` needs to establish the mapping between these two graphs, in order to support further

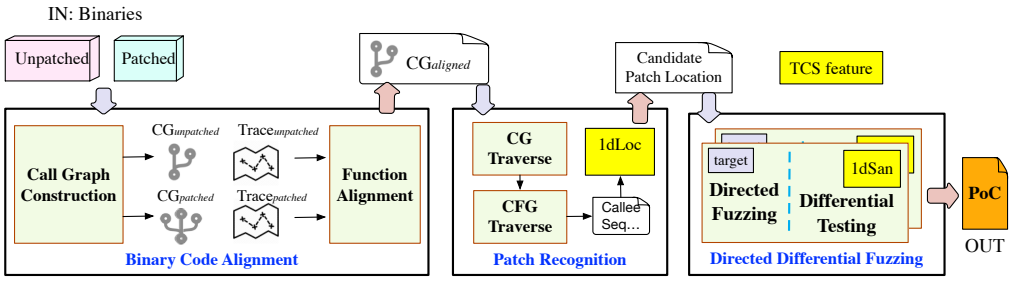


Figure 1: Overview of the workflow of our solution 1dFuzz.

TCS-related analysis or testing. We utilize the information collected during dynamic testing to perform alignment. The key observation is that if both the patched and unpatched binaries are tested with the same input, then most of the functions invoked along their traces *must* be similar, except for patch-related functions. In other words, the two traces have highly similar sequences of functions, which can be used to align subgraphs in the two call graphs. Thus, we could rely on traces to align functions in two binaries gradually.

Specifically, given the call graphs of the patched and unpatched binaries constructed in §4.2.1, 1dFuzz runs both binaries with the same input and records all function calls during the execution. 1dFuzz adds the pair of entry functions of the two binaries to an initially empty matched set. Then, 1dFuzz finds the next matching pair to add to the match set with a similarity scoring scheme. For every unmatched function f in the unpatched CG and every unmatched function f' in the patched CG, we calculate the similarity score of (f, f'), which consists of the following factors.

In-Out Factor: In and out degrees of function nodes. When two functions have similar in and out degrees in the call graph, they are more likely to match. 1dFuzz denotes the in-out degree factor as $\frac{\min(I_f, I_{f'})}{\max(I_f, I_{f'})} + \frac{\min(O_f, O_{f'})}{\max(O_f, O_{f'})}$, where I_f and $O_{f'}$ represents the in-degree of function f and the out-degree of function f'.

Operand Factor: Addressing mode of call instruction operands. There are multiple ways to store the indirect call target variable in memory at the call instruction, including stack variables and fields in nested objects, which have different addressing mode. Therefore, 1dFuzz records all types of function call parameters in the function. For the call instructions sharing the same addressing mode recorded during the disassembly stage, 1dFuzz denotes $\frac{\sum \min(\mathrm{MC}_{\mathrm{addr}}, \mathrm{MC}'_{\mathrm{addr}})}{\sum (\mathrm{MC}_{\mathrm{addr}} + \mathrm{MC}'_{\mathrm{addr}})/2}$ as the operand factor, where $\mathrm{MC}_{\mathrm{addr}}$ represents the number of function call parameters of a specific addressing mode in the unpatched CG and $\mathrm{MC}'_{\mathrm{addr}}$ in the patched.

Functions have reliable matching ancestors. 1dFuzz examines the caller functions of f and f' to see if there is a match. If f has its caller f_p and f' has f'_p, and (f_p, f'_p) is in the matched set, then 1dFuzz counts 1 point for the function pair, or otherwise, no points. In particular, if the calling relationship for (f_p, f) and (f'_p, f') both exists in the recorded function calling sequences, 1dFuzz additionally counts another 1 point for the function pair.

Reusable history matching result. The binary function alignment process runs iteratively, as new inputs are tested. Each round of the process will update the matching result. The result of a previous round can affect the confidence of the current round. 1dFuzz saves the result of the last round (empty for the first match). If the function f and function f' match in the previous round, 1dFuzz will count 0.5 points for the function pair.

The similarity score of the two functions summarizes these scoring factors. After scoring all possible function pairs, 1dFuzz selects the function pair with the highest matching score and adds the pair to the matched set. 1dFuzz then performs this operation repeatedly until one of the binaries has no remaining unmatched functions.

4.3 1dLoc: TCS-Based Patch Locator

1dFuzz needs to find candidate patches and set them as targets for directed fuzzing to explore. Note that more than 70% of the patches have the TCS feature. Thus, we propose to locate patches with TCS.

Since we only focus on patches with the TCS feature, we propose a TCS-based solution 1dLoc to locate candidate patch locations by analyzing the function call sequences in call graphs and finding differences between the call graphs of pre- and post-patch binaries. For instance, starting from two aligned functions, if they subsequently invoke different sequences of functions, it is likely that the aligned function is a candidate patched function.

Specifically, our solution 1dLoc first synchronously traverses pairs of aligned functions in both pre- and post-patch call graphs in a depth-first manner. For each pair of aligned functions, 1dLoc constructs the CFGs of each function from the binaries, then traverses all paths in the two CFGs, and collects the sequence of callee functions invoked on each path. We also collect callee functions of indirect calls if they are available at the call graph construction phase (with the help of dynamic testing). Therefore, we could get two sets of callee function sequences for each pair of aligned functions. Then we compare these two sets and check if they are different. If one set has a callee function sequence that is absent in the other set, we will mark the aligned function as a candidate patched function, which will be further explored by directed fuzzing.

4.4 1dSan: TCS-Based Sanitizer

We present a sanitizer 1dSan to catch PoCs that can trigger the patched vulnerability during dynamic testing. As discussed in §3, such a PoC often has the TCS feature, i.e., if fed with the PoC, the patched and unpatched binaries differ in the trailing part of their function call sequences because the patch alters the control flow. Therefore, we can record the function call sequences of the pre- and post-patch binaries during the testing, identify differences between the trailing parts, then reports a PoC if the TCS feature is present.

To test the pre-patch and post-patch binaries concurrently and check the TCS feature, we leverage the differential testing technique. As illustrated in Figure 2, we test two binaries with the same input and identical independent runtime environment. The

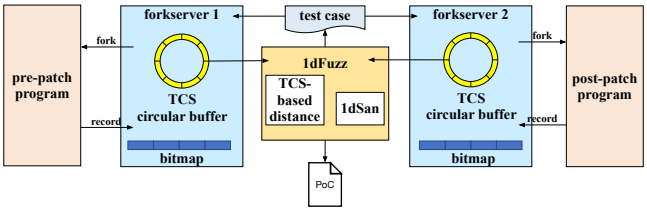


Figure 2: Directed differential testing of `1dFuzz`.

primary process launches two execution instances for the binaries, initializes both running environments using the same protocol, and distributes each round of test cases identically to both instances.

For efficiency, it does not record all callee functions during testing. Instead, it only records the last N functions called along the program execution. Specifically, it instruments the binary code to track function invocations and stores the callee information in a circular buffer shared with the fuzzer.

The sanitizer `1dSan` resides in the fuzzer rather than the instrumented binaries. It compares the two circular buffers after the test case finishes and reports the current test case as a PoC of a 1-day vulnerability if functions in these two buffers are different.

4.5 Directed Differential Fuzzing

Given the aligned pre-patch and post-patch binaries and candidate patch locations recognized by `1dLoc`, as well as the sanitizer `1dSan` to catch PoCs during differential testing, `1dFuzz` adopts a directed differential fuzzing solution to explore candidate patch locations.

4.5.1 Distance Metric for Directed Fuzzing. Traditional directed fuzzing solutions [10] utilize control flow distance to guide the fuzzer to get closer to the target. This distance metric is not optimal for exploring candidate patched functions recognized by `1dLoc`.

To determine the distance of the current test case to the target function, we first present the distance of a source function to the target function. Note that, given an edge (i.e., a caller to a callee) in the call graph, if the caller function has more call sites that will call the callee, then we believe the caller can enter the callee much easier. In other words, the distance between such caller and callee is shorter. Therefore, for each edge in the call graph, we will count how many call sites in the caller function will call the callee, and denote it as `CN` (> 0), then calculate the distance of this edge as $1 + \frac{1}{2 \times \texttt{CN}}$. Given the distance of every edge in the call graph, we will find the shortest path from the source function to the target function and get its distance.

Based on the distance from a source function to the target function, `1dFuzz` could determine the distance of a test case (i.e., its execution path) to the candidate patched function. Specifically, given the execution path and a target function, we iterate callee functions reversely in the path to find a function that is the *dominator* of the target function. In extreme cases, the dominator function is the first start function in the execution trace. Then, we take the distance from the dominator function to the target function as the distance to the target function of the test case.

4.5.2 Workflow of Directed Differential Fuzzing. Figure 2 demonstrates the core workflow of our solution. Conceptually, it applies a directed fuzzing and a differential testing at the same time.

We utilize differential testing to enable the sanitizer `1dSan` to catch PoCs for 1-day vulnerabilities. Specifically, the fuzzer generates a test case and tests the pre-patch and post-patch binaries, then `1dSan` compares the circular buffers of the two binaries after testing and reports a PoC if they differ.

Further, to explore the candidate patch locations recognized by `1dLoc`, we conduct a directed fuzzing during the differential testing. For each test case in the fuzzing queue, `1dFuzz` first calculates the distance between its execution path and the candidate patched function, then *selects test cases with shorter distances* for further mutation. If there are multiple candidates, `1dFuzz` selects the smallest distance among all candidates, takes that candidate as the target to explore later, and prioritizes the associated test case in the queue for further mutation. To avoid falling into local optimum, `1dFuzz` also randomly prioritizes other seeds with a smaller probability.

5 IMPLEMENTATION

Figure 3 shows the details of implementation, which has three major components. The *binary code alignment* component includes a static disassembly tool based on IDAPython, a dynamic trace recording tool based on Intel Pin [28], and an aligning and searching script developed in Python. The *patch recognition* `1dLoc` utilizes angr [39] to reconstruct CFG for functions and develops a Python script to locate the candidate patched functions. The *directed differential testing* module is implemented on AFL++ [20]. We modify the QEMU mode [4] of AFL++ to record the function call trace for `1dSan`, the CFG edge coverage, the call graph edge coverage, and the distance to candidate patched functions. Seeds with either new control flow edge coverage or function call edge coverage will be preserved in the seed queue. The length of the trailing call sequence (i.e., N) is set to 8. A smaller N will cause false positives, while a larger N will have higher overheads. We have tried several settings of N and found that 8 is a better value in practice. We also add a second fork server to AFL++ to run the two binaries simultaneously.

In addition, `1dFuzz` utilizes an iterative scheme to update the call graph. Therefore, `1dFuzz` also records the function call edges during the fuzzing process. When a new function call edge not present in the call graph is discovered, `1dFuzz` records the seed input and sends it back to the dynamic trace recording module. The dynamic analysis tool re-executes the patched and unpatched binaries with this input and adds the newly found edges to the call graph, as presented in §4.2.1. Then, the candidate patched functions are updated with the new call graphs and reloaded into the fuzzing module. `1dFuzz` further recalculates the distances for all seeds in the queue to guide fuzzing, since the call graph has changed.

6 EVALUATION

Our evaluation was designed to answer the following research questions in the experiments:

- **RQ1:** Can `1dFuzz` effectively reproduce PoCs for 1-day vulnerabilities, and how is it compared to state-of-the-art (SOTA)?
- **RQ2:** Ablation Study: How effective is the patch locator `1dLoc`?
- **RQ3:** Ablation Study: How effective is the patch sanitizer `1dSan`?
- **RQ4:** What is the impact of the version distance between pre-patch and post-patch binaries?
- **RQ5:** What is the performance of the static analysis module, in terms of time cost and precision?

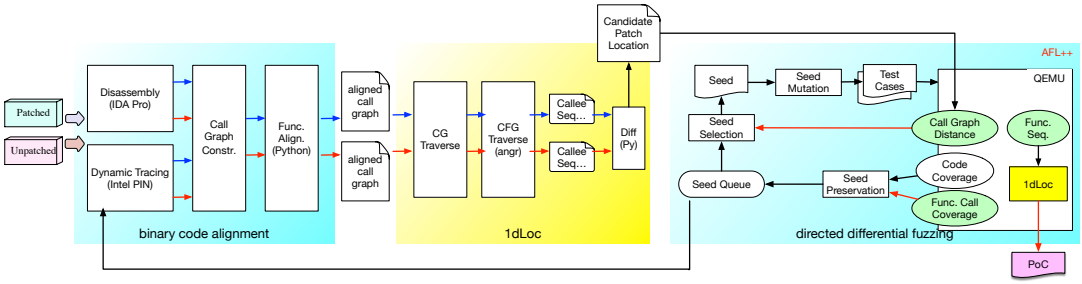


Figure 3: Implementation details of 1dFuzz.

Table 3: Projects and vulnerabilities in our evaluation.

Project	CVEs	Project	CVEs	Project	CVEs
curl	3	libarchive	8	openssl	11
FFmpeg	76	libav	3	openvpn	2
file	22	libpng	1	php-src	7
httpd	1	LibRaw	5	proftpd	1
ImageMagick	59	mysql-server	2	qpdf	1
jasper	20	openjpeg	16	zlib	4

6.1 Experiment Setup

Benchmarks of 1-day vulnerabilities. To obtain test cases for our evaluation, we complement our dataset by selecting projects collected in SecretPatch [47] that satisfy any of the following criteria: ① The project is a member of the top-10 vulnerable projects in SecretPatch. ② The project is widely used worldwide, including servers, readers, databases, and data transfer tools. ③ The project has been evaluated by related researches [20, 25, 50, 55].

In total, we get 18 project repositories (as shown in Table 3) which have 242 patches (and CVEs). For each vulnerability, we check out the code before and after the patch commit and build the pre- and post-patch binaries with default configurations.

SOTA directed fuzzing solution: Beacon. Since no 1-day vulnerability reproducing work is similar to 1dFuzz, we compare it with the most recent directed fuzzing solution, Beacon [25]. Note that Beacon accelerates directed fuzzing by pruning infeasible paths during fuzzing and *is orthogonal to our solution*, and thus we can rebase our solution to Beacon and get better performance. Since Beacon is not open source, we leave it a future work to port our solution atop it. This is not a fair comparison because (1) Beacon relies on source code to perform a dedicated static analysis, and (2) it relies on prior knowledge of where the target is. We conducted this experiment to show the effectiveness of 1dFuzz even compared to the SOTA solution, which works in the ideal scenario.

Baseline fuzzers. We collect two representative fuzzers as baselines, including a coverage-guided fuzzer AFL++, which integrates many state-of-the-art improvements, and an *open-source* directed fuzzer AFLGo. We migrate AFLGo to the latest AFL [1] and qemuafl, as the original version does not work in the QEMU mode in testing binary programs. We run the QEMU mode for AFL++ and 1dFuzz to directly test the binaries and run AFLGo following its default configuration listed in the document. 1dFuzz uses (1) 1dLoc as a directed target locator solution, (2) 1dSan as the patch sanitizer, and (3) a TCS-based distance metric. For control experiments, we developed a variant (i.e., AFLpp-d) of AFL++, which integrates the TCS-based distance metric (§4.5.1) for directed fuzzing, and developed fuzzer variants that use a trivial patch locator otLoc and the well-known binary sanitizer QASan. In total, we have 11 settings consisting of fuzzers of different locators, fuzzing engines, and sanitizers, labeled as ID 1 to 11 in Tables 4, 7, 8, and 9.

Running environment. Our evaluations are conducted on the machine with Intel(R) Xeon(R) Gold 6154 CPU @3.00GHz and 502 GB RAM, running Ubuntu 20.04 LTS. We build a separate Docker environment for every setting of fuzzer and every pair of pre- and post-patch binaries. For every docker instance, we assign 2 CPU cores to meet the requirement of differential testing. As for baseline fuzzers without differential testing, we assign each of their Docker instances with 3 CPUs: 2 of them are used to run two fuzzing processes, and the remaining one is idle to prevent Docker container resource scheduling problems and crashes.

In each experiment group, 1dFuzz and baseline fuzzers run for 24 hours, while Beacon *runs for 48 hours* to make a fair comparison, since 1dFuzz has two instances to perform differential testing. Each experiment is repeated 3 times to reduce errors. In total, we have spent more than 383,328 CPU hours in all experiments.

6.2 Effectiveness of 1dFuzz

To answer **RQ1**, we compare the performance of 1dFuzz at discovering 1-day vulnerabilities with the baseline and SOTA fuzzers.

6.2.1 Comparison with Baseline Fuzzers. In the first group, we compare 1dFuzz with two baselines: AFL++ and AFLGo. We have also evaluated AFL++ and AFLGo with extra patch locator and sanitizer support, which are presented in the following sections.

The results are listed in Table 4. 1dFuzz successfully triggers 95 vulnerabilities. Among them, 84 vulnerability patches are in the benchmark and have the TCS feature, 7 vulnerability patches are in the benchmark but have no TCS feature (not caught by 1dSan but instead crash the program), and 4 vulnerability patches are outside the benchmark. For comparison, AFL++ triggers bugs for 25 in-benchmark TCS patches and 2 others, and AFLGo triggers bugs for 31 in-benchmark TCS patches and 4 others. 1dFuzz finds 2.5x (=91/26-1) and 1.76x (=91/33-1) more in-benchmark bugs than AFL++ and AFLGo respectively, showing that 1dSan *is more effective at reproducing 1-day vulnerabilities than baselines.*

We also record the average time cost of the first PoC for each fuzzer. AFL++ takes an average of 7h48m to generate the first PoC, and AFLGo takes 6h52m. 1dFuzz takes only 4h27m, which is 1.75x and 1.54x faster than AFL++ and AFLGo, showing that 1dSan *is also more efficient at reproducing 1-day vulnerabilities than baselines.*

6.2.2 Comparison with SOTA Directed Fuzzing Solution: Beacon. Beacon relies on source code to perform analysis and cannot reproduce 1-day vulnerabilities for binary programs. It neither can recognize candidate patch locations. In addition, it is not open

Table 4: Fuzzing results of the Baseline Group, where we compare the performance of 1dFuzz to AFL++ and AFLGo.

Group	Solution				Patch	Target	Vulnerabilities						Average
	ID	Locator	Fuzzer	Sanitizer	Distance	Number	Total	ratio	iB+TCS	ratio	iB+nTCS	nB	Time
Baseline	1	/	AFLpp	/	c	242	**27**	0.28	**25**	0.30	1	1	**7h48m**
	2	/	AFLGo	/	c	242	35	0.37	31	0.37	2	2	6h52m
	3	**1dLoc**	**AFLpp-d**	**1dSan**	**c**	**242**	**95**	**1.00**	**84**	**1.00**	**7**	**4**	**4h27m**
Best Improvement							3.52x		3.36x				1.75x

Patch distance *c*: there is only one commit gap between the pre- and post-patch binaries. *iB*: crash hits target patch in benchmark. *nB*: newly found crash not in benchmark. *TCS*: crash conforms to the TCS feature. *nTCS*: otherwise. Average Time: average time to first generated PoC.

Table 5: Comparison with AFLGo and Beacon in terms of the time to trigger each vulnerability. T.O. indicates time out.

No.	Program	CVE	AFLGo (48h)	Beacon (48h)	1dFuzz (24h)
1	cxxfilt	2016-4491	7.74h	1.38h	1.74h
2		2016-6131	5.88h	0.84h	1.09h
3	objdump	2017-8392	T.O.	8.42h	13.41h
4		2017-8396	T.O.	39.03h	T.O.
5		2017-8398	T.O.	40.51h	T.O.
6		2017-16828	T.O.	22.24h	T.O.
7		2018-17360	T.O.	45.69h	T.O.
8	objcopy	2017-7303	T.O.	20.09h	T.O.
9		2017-8393	T.O.	19.78h	T.O.
10		2017-8394	T.O.	4.46h	5.65h
11		2017-8395	T.O.	3.83h	6.82h
12	cjpeg	2018-14498	T.O.	11.46h	14.94h
13		2020-13790	7.34h	3.98h	5.24h
14	swftophp	2016-9827	1.25h	0.31h	0.39h
15		2016-9829	T.O.	5.54h	6.18h
16		2016-9831	2.52h	0.62h	1.48h
17		2017-7578	2.43h	0.29h	1.14h
18		2017-9988	T.O.	1.45h	5.7h
19		2017-11728	T.O.	11.14h	T.O.
20		2017-11729	4.34h	1.02h	2.27h
21		2018-7868	T.O.	1.75h	4.25h
22		2018-8807	10.71h	1.89h	5.28h
23		2018-8962	T.O.	1.92h	3.37h
24		2018-11095	T.O.	3.13h	3.61h
25		2018-11225	T.O.	2.84h	4.00h
26		2018-11226	T.O.	3.98h	4.82h
27		2018-20427	T.O.	3.14h	8.2h
28		2019-9114	T.O.	3.53h	3.72h
29		2019-12982	T.O.	2.47h	2.63h
30		2020-6628	T.O.	3.91h	5.22h
31	xmllint	2017-5969	2.07h	0.17h	0.29h
32		2017-9047	T.O.	16.55h	T.O.
33		2017-9048	T.O.	18.00h	T.O.
34		2017-9049	T.O.	31.56h	T.O.
35	lrzip	2017-8846	5.05h	1.78h	2.3h
36		2018-11496	3.01h	1.17h	1.85h
37	pngimage	2018-13785	T.O.	3.22h	2.91h
38	avaconv	2018-18829	T.O.	47.97h	T.O.
Avg.			**2.09x**	**0.60x**	**1.00x**

source. Therefore, we take the data from Beacon's paper, and take all 38 vulnerabilities that Beacon has successfully triggered within 48 hours (i.e., the time budget in our experiments) as the dataset.

Table 5 shows the results. Among these 38 vulnerabilities that Beacon triggers within 48 hours, 1dFuzz successfully reproduces 27 within 24 hours, while AFLGo only reproduces 11 within 48 hours. Compared to 1dFuzz, AFLGo on average takes 2.09x time to trigger the vulnerabilities, while Beacon only takes 0.6x.

Result analysis. 1dFuzz significantly outperforms AFLGo, which has been analyzed in §6.2.1, but is worse than Beacon. This result meets our expectation, due to the following reasons. First of all, Beacon relies on source code of target applications to perform a dedicated static analysis to support infeasible paths pruning, while 1dFuzz and AFLGo work for binary programs and are more practical. This static analysis scheme is not easy to port to binary programs. Second, the static analysis is time consuming, but the overhead is not counted in the results. Last but not the least, both Beacon and AFLGo cannot recognize patch locations or catch runtime patch triggering behaviors. In our experiment, we manually specify the unique location of each target vulnerability for Beacon and AFLGo, which saves them from the trouble of two challenging tasks: recognizing patches and catching PoCs. On the other hand, 1dFuzz automatically recognizes a set of candidate patch locations and catch potential PoCs at runtime, inevitably has time cost. Note that, due to the lack of prior knowledge of the patch location, 1dFuzz makes tradeoffs between multiple candidate targets, which also slows down the performance.

Table 6: False Positive Testing Results using Magma.

Project	# Patch	# TCS PoC	# nTCS PoC	# T.O.
libpng	7	3	0	4
libtiff	14	6	0	8
libxml2	17	6	0	11
poppler	22	7	0	15
openssl	20	5	0	15
Total	80	27	0	53

TCS: PoC triggers the vulnerability logging. *nTCS*: otherwise. *T.O.*: time out.

6.2.3 False Positive Verification: Magma. 1dFuzz reports PoCs efficiently by detecting runtime differences in TCS features via 1dSan. In order to further verify that these PoCs trigger 1-day vulnerabilities rather than false alarms, we design an additional experiment.

Magma [24], one of the fuzzing benchmarks, consists of multiple vulnerabilities, each having two macro-controlled changes to the original project source code: one inserts vulnerability-triggering assertions for logging, and the other fixes the vulnerability with a patch. Based on this, we build a test set by enabling these macros, respectively, with the logging version as the pre-patch binary and the fixed version as the post-patch binary. The PoC generated by 1dFuzz is then verified on the pre-patch logging binary to check if the vulnerability is triggered. We tested 80 vulnerabilities from the first five projects of Magma.

The results are shown in Table 6. For the 80 patches, 1dFuzz reproduced 27 PoCs within 24 hours, and all of them successfully triggered the vulnerability, i.e., there were no false positives. The other 53 targets timed out. It is worth noting that, this experiment is not perfect. A more comprehensive benchmark that has not only security patches but also functionality updates is required. Though, in order to evaluate false alarms, abundant manual efforts are needed to build the ground-truth results of vulnerabilities for this benchmark. We leave it as a future work.

6.3 Ablation Study: Patch Locator

To answer **RQ2** and evaluate of the 1dLoc module, which finds candidate patch locations, we designed the control experiment of Group 1dLoc of directed fuzzing.

Table 7: Fuzzing results of the 1dLoc Group, where we compare 1dLoc with another patch locator.

Group	Solution				Patch	Target	Crash						Average
	ID	Locator	Fuzzer	Sanitizer	Distance	Number	Total	ratio	iB+TCS	ratio	iB+nTCS	nB	Time
1dLoc	4	/	AFLpp	1dSan	c	242	**36**	0.38	**34**	0.40	0	2	**6h12m**
	3	**1dLoc**	**AFLpp-d**	**1dSan**	**c**	**242**	**95**	**1.00**	**84**	**1.00**	**7**	**4**	**4h27m**
	5	otLoc	AFLpp-d	1dSan	c	242	67	0.71	61	0.73	5	1	4h49m
	6	1dLoc	AFLGo	1dSan	c	242	44	0.46	39	0.46	3	2	5h22m
	7	otLoc	AFLGo	1dSan	c	242	39	0.41	35	0.42	3	1	5h48m
Best Improvement							2.64x		2.47x				1.39x

Patch distance *c*: there is only one commit gap between the pre- and post-patch binaries. *iB*: crash hits target patch in benchmark. *nB*: newly found crash not in benchmark. *TCS*: crash conforms to the TCS feature. *nTCS*: otherwise. Average Time: average time to first generated PoC.

Table 8: Fuzzing results of the 1dSan Group, where we compare 1dSan with another sanitizer.

Group	Solution				Patch	Target	Vulnerabilities						Average
	ID	Locator	Fuzzer	Sanitizer	Distance	Number	Total	ratio	iB+TCS	ratio	iB+nTCS	nB	Time
1dSan	8	1dLoc	AFLpp-d	/	c	242	69	0.73	63	0.75	6	3	5h17m
	9	1dLoc	AFLpp-d	QASan	c	242	72	0.76	60	0.71	7	5	5h51m
	3	**1dLoc**	**AFLpp-d**	**1dSan**	**c**	**242**	**95**	**1.00**	**84**	**1.00**	**7**	**4**	**4h27m**
	6	1dLoc	AFLGo	1dSan	c	242	44	0.46	39	0.46	3	2	5h22m
	10	1dLoc	AFLGo	QASan	c	242	**38**	0.40	**31**	0.37	5	2	**6h09m**
Best Improvement							2.50x		2.71x				1.38x

Patch distance *c*: there is only one commit gap between the pre- and post-patch binaries. *iB*: crash hits target patch in benchmark. *nB*: newly found crash not in benchmark. *TCS*: crash conforms to the TCS feature. *nTCS*: otherwise. Average Time: average time to first generated PoC.

Experiment setup. All solutions use the same sanitizer 1dSan but different patch locators and fuzzing engines. To compare with 1dLoc, we present a traditional patch locator otLoc which relies on binary diffing. It uses BinExport [3] to obtain function-level differences between the pre- and post-patch binaries identified by BinDiff, and selects candidate patched functions as targets for directed fuzzing. Solution 4 uses no patch locators and only performs coverage-guided fuzzing rather than directed fuzzing. Solution 5 uses otLoc as the patch locator. Solutions 6 and 7 try to use AFLGo as the alternative fuzzing engine with 1dLoc and otLoc as patch locators, respectively.

Results of PoC generation. The results are shown in Table 7. Results show that the presence of 1dLoc will significantly promote the effectiveness of fuzzers. Due to the lack of a directed target locator, Solution 4 significantly lags behind other solutions in the total number of reproduced PoCs (only 38% of 1dFuzz), the number of in-benchmark TCS patches (40% of 1dFuzz), and the average generation time of first reproduced PoC (1.39x more time cost).

We can also learn that the fuzzer engine AFL++ outperforms AFLGo in terms of both vulnerabilities reproducing and time cost, by comparing Solution 3 vs. 6 and Solution 5 vs. 7. Furthermore, our patch locator 1dLoc has better performance than the trivial locator otLoc by comparing Solution 3 vs. 5, and Solution 7 vs. 6, showing that *our patch locator has better efficiency and effectiveness.*

6.4 Ablation Study: Vulnerability Sanitizer

To answer **RQ3** and evaluate the 1dSan module, we design the control experiment of Group 1dSan.

Experiment setup. All fuzzing solutions use the same patch locator, 1dLoc, but different sanitizers and fuzzing engines. To compare with 1dSan, we utilize a SOTA binary sanitizer QASan. Solution 8 does not use any sanitizer. Solution 9 utilizes the sanitizer QASan integrated with AFL++. Solutions 6 and 10 change the fuzzing framework to AFLGo while using 1dSan and QASan. We migrate QASan to the latest qemuafl used for AFLGo, as explained in §6.1. Solution 6 has already been evaluated in the 1dLoc Group, so we directly copy its results.

Results of PoC generation. The results are shown in Table 8. By comparing Solutions 3 to 8, we find that the absence of 1dSan leads to a 30% decrease in PoCs generation. Meanwhile, there is no significant influence brought by QASan (Solutions 9 vs 8). Further, the fuzzer engine AFLpp-d significantly outperforms the engine AFLgo by comparing Solution 3 to 6 and Solution 9 to 10. Lastly, we can see that our sanitizer 1dSan also greatly outperforms QASan, by comparing Solution 3 to 9 and Solution 6 to 10, showing that *our sanitizer has better efficiency and effectiveness.*

6.5 Robustness of Patch Version Distance

In the above experiments, the pre-patch and post-patch binaries have limited differences, as they are taken from two successive commits. However, in practice, software vendors will not always release new binaries after a new commit. In contrast, they are more likely to release a collection of updates fixing vulnerabilities found in a period. In addition to security updates, the update packages also include functional updates. In these cases, there may be significant differences between the pre- and post-patch binaries, bringing extra challenges to 1-day vulnerabilities reproducing. Therefore, we introduce another group of experiments to demonstrate the efficiency of 1dFuzz in considering patch distance for **RQ4**.

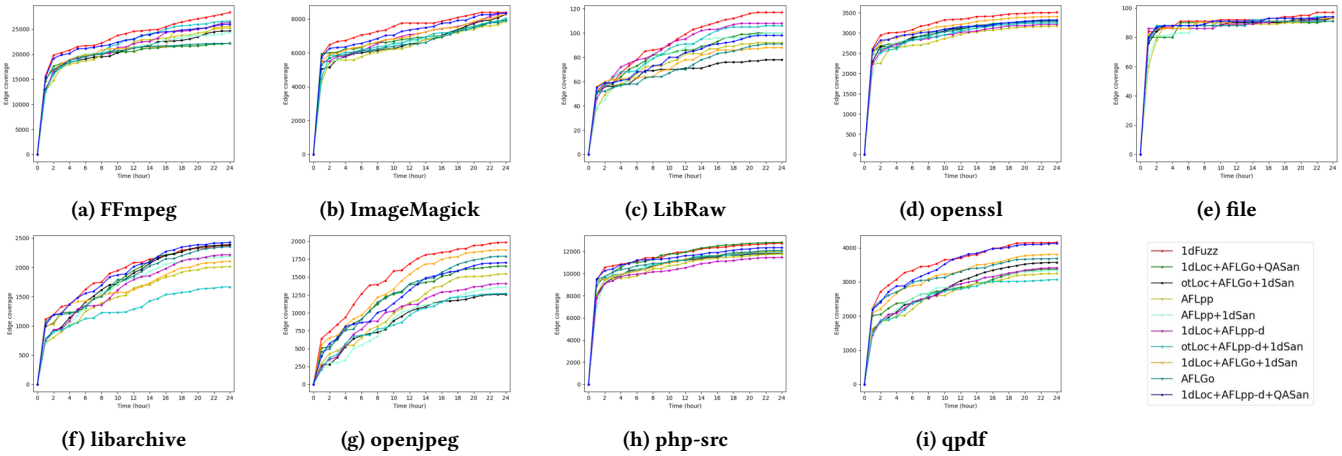
Experiment setup. The version number used by open-source software usually follows the semantic versioning scheme, i.e., the Major.Minor.Patch semantics. The major number represents breaking changes when there is a significant software upgrade. The minor number corresponds to non-breaking changes, including small patches and adjustments that bump the patch number. We believe the two software versions with adjacent minor numbers before and after the patch contain noise patches against the security patches, so we test 1dFuzz on such binary pairs in this group. When rebuilding the binary benchmark, we retrieve the version numbers of the security patch commit, checkout source code of adjacent minor versions, then compile them into pre-patch and post-patch binaries. Other configurations remain the same.

Results of PoC generation. The results are shown in Table 9. Compared to single commit patch distance, the number of PoCs

Table 9: Fuzzing results of the Distance Group, where we change the distance between pre- and post-patch binaries.

Group	Solution				Patch	Target	Crash						Average
	ID	Locator	Fuzzer	Sanitizer	Distance	Number	Total	ratio	iB+TCS	ratio	iB+nTCS	nB	Time
Distance	11	1dLoc	AFLpp-d	1dSan	m	242	**48**	0.51	**40**	0.48	4	4	**7h33m**
	3	**1dLoc**	**AFLpp-d**	**1dSan**	**c**	**242**	**95**	**1.00**	**84**	**1.00**	**7**	**4**	**4h27m**
Best Improvement							1.98x		2.10x				1.70x

Patch distance *c*: only one commit gap between the pre- and post-patch binaries, *m*: one minor version gap. *iB*: crash hits patch in benchmark. *nB*: newly found crash not in benchmark. *TCS*: crash conforms to the TCS feature. *nTCS*: otherwise. Average Time: average time to first PoC.



(a) FFmpeg (b) ImageMagick (c) LibRaw (d) openssl (e) file

(f) libarchive (g) openjpeg (h) php-src (i) qpdf

Figure 4: Edge coverage for different settings of fuzzers on half (odd ones in alphabetical order) of the 18 projects in 24h.

Table 10: Average edge coverage of all 11 fuzzers in 24h.

Group	Baseline			1dLoc				1dSan			Distance
Solution	1	2	3	4	5	6	7	8	9	10	11
curl	**4419**	3836	4345	4163	4253	4273	4197	4395	4338	3985	3876
FFmpeg	25198	22171	**28331**	24367	26630	25552	24629	26312	25973	22203	20895
file	93	91	**97**	91	94	93	93	93	94	91	90
httpd	3874	4443	**5383**	4335	3973	5460	4565	4010	4273	5035	2794
ImageMagick	7847	7918	**8385**	7889	8035	8269	8267	8365	8328	7921	6466
jasper	6011	6154	**7183**	6299	6318	6333	6252	6228	7115	6254	5784
libarchive	2012	2355	**2372**	2185	1665	2105	2388	2215	2427	2386	2193
libav	2210	2264	**2825**	2251	2544	2599	2205	2123	2586	2485	1640
libpng	658	733	**845**	696	656	726	659	653	770	802	640
LibRaw	92	91	**117**	100	106	88	78	108	98	100	84
mysql-server	12768	13290	**14480**	9760	12678	11592	13219	13127	13859	12283	12796
openjpeg	1545	1790	**1984**	1360	1271	1880	1258	1407	1701	1654	1074
openssl	3169	3290	**3507**	3291	3247	3406	3290	3212	3321	3306	3010
openvpn	8164	10065	**11620**	8253	8233	9528	9384	8843	11173	9740	8301
php-src	11773	12089	**12735**	11978	11954	12026	11819	11463	12343	12816	11202
proftpd	855	988	**1071**	871	937	1068	849	987	1016	914	797
qpdf	3250	3685	**4156**	3337	3067	3809	3574	3411	4125	3376	3124
zlib	149	180	**195**	171	170	171	189	166	194	160	167

ID number in the Solution row represents 11 groups of settings for fuzzers in Tables 4, 7, 8 and 9.

successfully reproduced by 1dFuzz on the minor version distance benchmark is only 0.51x, and in-benchmark TCS-conformed PoCs are only 0.48x. This indicates that, as the patch distance increases, 1dFuzz can still reproduce the vulnerabilities according to the TCS features. Nevertheless, it inevitably has lower performance due to noises in the binaries, including non-security functional updates, code refactoring, and compilation optimizations that lead to additional changes to the target binary.

6.6 Performance on Code Coverage

Table 10 shows the average edge coverage of all fuzzing solutions we evaluate on all targets in the benchmark, as half are plotted in Figure 4, due to space limitations. We found that the edge coverage curve of 1dFuzz is at the top during the 24-hour fuzzing process in all programs. In other words, 1dFuzz can reach a relatively high edge coverage early in fuzzing and stay ahead.

Results of the Baseline Group. We can find that the edge coverage achieved by 1dFuzz (ID 3) after 24 hours of fuzzing exceeds the results of AFL++ (ID 1) and AFLGo (ID 2) in all cases, except that *curl*'s is lower than AFL++. In 6 programs, 1dFuzz can find 1,000 more new edges than AFL++.

Results of the 1dLoc Group. The edge coverage of 1dFuzz (solution 3) exceeds that of solutions 4-7 in most programs. Furthermore, solutions using 1dLoc have relatively higher edge coverage at the beginning than those using otLoc.

Results of the 1dSan Group. Compared with the solutions 8-10 under 1dSan that do not use 1dSan, our solution 1dFuzz (solution 3) has better code coverage on 15 out of the 18 programs, except for *curl*, *libarchive*, and *php-src* where the gap is below 2.4%.

Results of the Distance Group. The edge coverage of minor version patch distance (Solution 11) lags behind the single commit patch distance 1dFuzz (Solution 3) across all programs in the benchmark. On *FFmpeg*, *mysql-server*, and *openvpn* programs, Solution 3 discovers 1,000 more new edges than Solution 11.

Result analysis. In summary, the results on code coverage show that (1) 1dFuzz outperforms baseline fuzzers, (2) fuzzers with 1dLoc outperform fuzzers with otLoc, and (3) fuzzers with 1dSan outperform fuzzers with QASan. Although 1dFuzz is not designed towards increasing the code coverage, this result is reasonable. During fuzzing, seeds with either new control flow edge coverage or function call edge coverage will be preserved in the seed queue by 1dFuzz. 1dLoc can identify candidate patches with higher accuracy, leading the fuzzer to explore more code with differences in pre- and post-patch binaries, and thus increase the code coverage. 1dSan

Table 11: 1dLoc and otLoc accuracy and performance.

Project	# CVE	# Func	1dLoc					otLoc		
			Time	TP	FN	TP24	FN24	Time	TP	FN
curl	3	2	4.2	1	1	1	1	7.2	1	1
FFmpeg	76	87	129.8	60	27	65	22	142.0	52	35
file	22	29	3.9	18	11	21	8	5.2	18	11
httpd	1	1	9.6	1	0	1	0	13.4	1	0
ImageMagick	59	74	62.7	57	17	63	11	90.1	51	23
jasper	20	23	4.9	15	8	15	8	7.5	12	11
libarchive	8	8	9.2	6	2	6	2	14.5	6	2
libav2	3	3	178.3	2	1	2	1	190.3	2	1
libpng	1	1	1.7	0	1	1	0	2.3	1	0
LibRaw	5	13	1.6	8	5	10	3	2.1	9	4
mysql-server	2	15	14.8	13	2	13	2	16.9	10	5
openjpeg	16	19	2.2	14	5	15	4	3.0	12	7
openssl	11	12	41.8	9	3	11	1	47.4	8	4
openvpn	2	2	17.2	2	0	2	0	20.8	2	0
php-src	7	9	112.5	3	6	6	3	169.2	4	5
proftpd	1	1	8.2	1	0	1	0	10.8	1	0
qpdf	1	5	24.7	3	2	4	1	30.4	3	2
zlib	4	2	2.6	2	0	2	0	3.4	1	1

CVE #: CVEs of software. *Func #*: manually confirmed TCS affected function. *Time*: analysis time cost. *TP / TP24*: identified functions initially / after 24 hours of fuzzing. *FN / FN24*: not identified functions.

can help catch more PoCs for 1-day vulnerabilities, which will be preserved in the seed queue and improve the efficiency of the fuzzer.

6.7 Performance of Static Analysis

In order to answer **RQ5**, we measure the performance and accuracy of the patch location analysis component 1dLoc.

Experiment setup. For each project in the benchmark, we count the number of CVEs and the number of ground-truth functions related to the TCS feature. Then, we manually verify the results of candidate patched functions recognized by 1dLoc and otLoc and count the true positives (TP) and false negatives (FN).

The patch locator 1dLoc locates candidate patched functions via two phases. Firstly, a pure static analysis result is yielded before the fuzzing process. Then, in the fuzzing process, the indirect call edges will be updated continuously so that 1dLoc will update the candidate patched functions continuously. Thus, we record two numbers of candidate patched functions, one before the fuzzing starts (TP and FN) and one after 24 hours of fuzzing (TP24 and FN24). We also record the average time cost of each patch locator.

Results. As shown in Table 11, in general, 1dLoc takes less analysis time than otLoc. This is because the selected otLoc does extra operations with IDA files, which brings extra overhead. Regarding analysis accuracy, in most cases, 1dLoc correctly identifies more functions than otLoc. Only for tests on *linpng*, *libraw*, and *php-src*, otLoc performs a little better than 1dLoc. When comparing *TP* with *TP24*, we can see that, after 24 hours of fuzzing, 1dLoc could recognize more candidate patched functions with the TCS feature for 8 of 18 projects. It further confirms that *our strategy of updating call graphs with fuzzing data is effective and useful.*

7 DISCUSSIONS

1dFuzz has shown the effectiveness of TCS features in reproducing 1-day vulnerability PoCs. However, 1dFuzz also has some shortcomings, which we leave as future work to address.

C/C++ binaries. 1dFuzz currently focuses on 1-day vulnerability reproduction for binary programs and thus builds testing benchmarks primarily from the C/C++ family of programs dataset. In the next step, we will consider more types of programs, such as Java and Python on virtual machines and interpreters, to find reliable methods to reproduce 1-day vulnerabilities in those programs.

TCS feature. The trailing call sequence feature characterizes changes in the trailing part of function call traces between patched and unpatched binary programs. This feature has been proven useful by 1dFuzz in reproducing 1-day vulnerabilities from security patches. However, it inevitably has false positives, since non-security patches may also have the TCS feature. But PoCs for these non-security patches are also useful, since they can be used as test cases for regression testing or fuzzing the newly created code. On the other hand, we will design a more fine-grained metric in the future, which takes features such as memory access into consideration, to characterize the feature of security patches.

Improving the general fuzzing framework. 1dFuzz mainly focuses on guiding directed differential fuzzing with TCS features and does not make many improvements to the fuzzing framework itself, as it is orthogonal to the core contribution of this paper. We will incorporate outstanding optimizing techniques from other recent works, such as pruning from Beacon, to further improve the performance and efficiency of 1dFuzz in the future.

8 THREAT TO VALIDITY

1dFuzz aims at reproducing 1-day vulnerabilities for binary programs, but there are many open challenges in binary analysis, which may cause threats to the validity.

Binary disassembly. In our prototype, we rely on a commercial tool IDA Pro to disassemble binaries. However, it may have false positives and false negatives in function recognition, which will cause problems for further TCS-related analysis.

Indirect call recognition. TCS analysis relies on the call graph, whose construction heavily relies on recognizing targets of indirect calls. However, it is extremely difficult to recognize the potential targets of indirect calls. Any mistakes in the call graph recognition would cause 1dFuzz to have false positives or false negatives in identifying potential patches. Note that, there is an AI-based solution Callee [53] recently accepted by IEEE S&P 2023, which could greatly help mitigate this threat. We leave it as a future work to adopt this solution.

9 CONCLUSION

1-day vulnerabilities pose more threats to end users in practice. Reproducing 1-day vulnerabilities is crucial for both adversaries and defenders. A core question to answer is what is the unique feature of a patch. We have conducted an empirical study and learned that the TCS is a common and unique feature for most patches. Based on this feature, we propose a novel directed differential testing solution 1dFuzz to reproduce 1-day vulnerabilities efficiently. Evaluation results showed that this solution is efficient and effective, significantly outperforming existing fuzzers.

ACKNOWLEDGMENTS

We sincerely thank all the anonymous reviewers for their valuable feedback that greatly helped us to improve this paper. This work is supported in part by the National Key Research and Development Program of China (2021YFB2701000), the National Natural Science Foundation of China (61972224, 62272265), and the Beijing National Research Center for Information Science and Technology (BNRist) under Grant BNR2022RC01006.

REFERENCES

[1] 2013. american fuzzy lop. https://lcamtuf.coredump.cx/afl/

[2] 2021. BinDiff. https://www.zynamics.com/bindiff.html

[3] 2021. BinExport. https://github.com/google/binexport

[4] 2021. qemuafl. https://github.com/AFLplusplus/qemuafl

[5] 2023. Hardware-assisted AddressSanitizer Design Documentation. https://clang.llvm.org/docs/HardwareAssistedAddressSanitizerDesign.html

[6] 2023. UndefinedBehaviorSanitizer. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html

[7] Nikolaos Alexopoulos, Manuel Brack, Jan Philipp Wagner, Tim Grube, and Max Mühlhäuser. 2022. How Long Do Vulnerabilities Live in the Code? A Large-Scale Empirical Measurement Study on FOSS Vulnerability Lifetimes. In *Proceedings of the 31st USENIX Conference on Security Symposium (SEC'22)*. USENIX Association, Boston, MA, USA, 359–376. https://www.usenix.org/conference/usenixsecurity22/presentation/alexopoulos

[8] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Kruegel, and Engin Kirda. 2009. Scalable, Behavior-Based Malware Clustering. In *Proceedings of the 2009 Network and Distributed System Security Symposium (NDSS'09)*. The Internet Society, San Diego, CA, USA, 18. https://www.ndss-symposium.org/ndss2009/scalable-behavior-based-malware-clustering/

[9] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. 2008. Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy (SP'08)*. IEEE, Oakland, CA, USA, 143–157. https://doi.org/10.1109/SP.2008.17 ISSN: 1081-6011.

[10] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS'17)*. Association for Computing Machinery, Dallas Texas USA, 2329–2344. https://doi.org/10.1145/3133956.3134020

[11] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based Greybox Fuzzing as Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS'16)*. Association for Computing Machinery, Vienna, Austria, 1032–1043. https://doi.org/10.1145/2976749.2978428

[12] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. 2018. Hawkeye: Towards a Desired Directed Grey-box Fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS'18)*. Association for Computing Machinery, Toronto, Canada, 2095–2108. https://doi.org/10.1145/3243734.3243849

[13] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP'18)*. IEEE, San Francisco, CA, USA, 711–725. https://doi.org/10.1109/SP.2018.00046

[14] Yaniv David, Nimrod Partush, and Eran Yahav. 2016. Statistical Similarity of Binaries. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'16)*. Association for Computing Machinery, Santa Barbara, CA, USA, 266–280. https://doi.org/doi.org/10.1145/2908080.2908126

[15] Steven H. H. Ding, Benjamin C. M. Fung, and Philippe Charland. 2019. Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP'19)*. IEEE, San Francisco, CA, USA, 472–489. https://doi.org/10.1109/SP.2019.00003

[16] Peter Dinges and Gul Agha. 2014. Targeted Test Input Generation Using Symbolic-concrete Backward Execution. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE'14)*. Association for Computing Machinery, Vasteras, Sweden, 31–36. https://doi.org/10.1145/2642937.2642951

[17] Yue Duan, Xuezixiang Li, Jinghan Wang, and Heng Yin. 2020. DeepBinDiff: Learning Program-Wide Code Representations for Binary Diffing. In *Proceedings of the 2020 Network and Distributed System Security Symposium (NDSS'20)*. The Internet Society, San Diego, CA, USA, 16. https://doi.org/10.14722/ndss.2020.24311

[18] Manuel Egele, Maverick Woo, and Peter Chapman. 2014. Blanket Execution: Dynamic Similarity Testing for Program Binaries and Components. In *Proceedings of the 23rd USENIX Conference on Security Symposium (SEC'14)*. USENIX Association, San Diego, CA, USA, 303–317. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/egele

[19] Andrea Fioraldi, Daniele Cono D'Elia, and Leonardo Querzoni. 2020. Fuzzing Binaries for Memory Safety Errors with QASan. In *2020 IEEE Secure Development (SecDev'20)*. IEEE, Atlanta, GA, USA, 23–30. https://doi.org/10.1109/SecDev45635.2020.00019

[20] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *Proceedings of the 14th USENIX Conference on Offensive Technologies (WOOT'20)*. USENIX Association, Berkeley, CA, USA, 12. https://doi.org/10.5555/3488877.3488887

[21] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. CollAFL: Path Sensitive Fuzzing. In *Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP'18)*. IEEE, San Francisco, CA, USA, 679–696. https://doi.org/10.1109/SP.2018.00040

[22] Debin Gao, Michael K. Reiter, and Dawn Song. 2008. BinHunt: Automatically Finding Semantic Differences in Binary Programs. In *Information and Communications Security*, Liqun Chen, Mark D. Ryan, and Guilin Wang (Eds.). Vol. 5308. Springer Berlin Heidelberg, Berlin, Heidelberg, 238–255. https://doi.org/10.1007/978-3-540-88625-9_16 Series Title: Lecture Notes in Computer Science.

[23] Jian Gao, Xin Yang, Ying Fu, Yu Jiang, Heyuan Shi, and Jiaguang Sun. 2018. VulSeeker-pro: Enhanced Semantic Learning Based Binary Vulnerability Seeker with Emulation. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE'18)*. Association for Computing Machinery, Lake Buena Vista, FL, USA, 803–808. https://doi.org/10.1145/3236024.3275524

[24] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. Magma: A Ground-Truth Fuzzing Benchmark. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 4, 3 (Dec. 2020), 1–29. https://doi.org/10.1145/3428334

[25] Heqing Huang, Yiyuan Guo, Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. 2022. BEACON: Directed Grey-Box Fuzzing with Provable Path Pruning. In *Proceedings of the 2022 IEEE Symposium on Security and Privacy (SP'22)*. IEEE, San Francisco, CA, USA, 36–50. https://doi.org/10.1109/SP46214.2022.9833751

[26] Gwangmu Lee, Woochul Shim, and Byoungyoung Lee. 2021. Constraint-guided Directed Greybox Fuzzing. In *Proceedings of the 30th USENIX Security Symposium (SEC'21)*. USENIX Association, Virtual Event, 3559–3576. https://www.usenix.org/conference/usenixsecurity21/presentation/lee-gwangmu

[27] Caroline Lemieux and Koushik Sen. 2018. FairFuzz: a targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE'18)*. Association for Computing Machinery, Montpellier France, 475–485. https://doi.org/10.1145/3238147.3238176

[28] Osnat Levi. 2021. Pin - A Dynamic Binary Instrumentation Tool. https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html

[29] Bingchang Liu, Wei Huo, Chao Zhang, Wenchao Li, Feng Li, Aihua Piao, and Wei Zou. 2018. αDiff: Cross-version Binary Code Similarity Detection with DNN. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE'18)*. Association for Computing Machinery, Montpellier France, 667–678. https://doi.org/10.1145/3238147.3238199

[30] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2014. Semantics-based Obfuscation-resilient Binary Code Similarity Comparison with Applications to Software Plagiarism Detection. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'14)*. Association for Computing Machinery, Hong Kong, China, 389–400. https://doi.org/10.1145/2635868.2635900

[31] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks. 2011. Directed Symbolic Execution. In *Static Analysis*, Eran Yahav (Ed.). Vol. 6887. Springer Berlin Heidelberg, Berlin, Heidelberg, 95–111. https://doi.org/10.1007/978-3-642-23702-7_11 Series Title: Lecture Notes in Computer Science.

[32] Jiang Ming, Dongpeng Xu, Yufei Jiang, and Dinghao Wu. 2017. BinSim: Trace-based Semantic Binary Diffing via System Call Sliced Segment Equivalence Checking. In *Proceedings of the 26th USENIX Security Symposium (SEC'17)*. USENIX Association, Vancouver, BC, Canada, 253–270. https://doi.org/10.5555/3241189.3241211

[33] Marc Ohm, Arnold Sykosch, and Michael Meier. 2020. Towards Detection of Software Supply Chain Attacks by Forensic Artifacts. In *Proceedings of the 15th International Conference on Availability, Reliability and Security (ARES'20)*. Association for Computing Machinery, Virtual Event, Ireland, 1–6. https://doi.org/10.1145/3407023.3409183

[34] Jiaqi Peng, Feng Li, Bingchang Liu, Lili Xu, Binghong Liu, Kai Chen, and Wei Huo. 2019. 1dVul: Discovering 1-Day Vulnerabilities through Binary Patches. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'19)*. IEEE, Portland, OR, USA, 605–616. https://doi.org/10.1109/DSN.2019.00066

[35] Theofilos Petsios, Adrian Tang, Salvatore Stolfo, Angelos D. Keromytis, and Suman Jana. 2017. NEZHA: Efficient Domain-Independent Differential Testing. In *Proceedings of the 2017 IEEE Symposium on Security and Privacy (SP'17)*. IEEE, San Jose, CA, USA, 615–632. https://doi.org/10.1109/SP.2017.27

[36] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference (ATC'12)*. USENIX Association, Boston, MA, USA, 309–318. https://doi.org/10.5555/2342821.2342849

[37] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications (WBIA'09)*. ACM Press, New York, NY, USA, 62–71. https://doi.org/10.1145/1791194.1791203

[38] Muhammad Shahzad, Muhammad Zubair Shafiq, and Alex X. Liu. 2012. A Large Scale Exploratory Analysis of Software Vulnerability Life Cycles. In *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*. IEEE, Zurich, 771–781. https://doi.org/10.1109/ICSE.2012.6227141

[39] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy (SP'16)*. IEEE, San Jose, CA, USA, 138–157. https://doi.org/10.1109/SP.2016.17

[40] Evgeniy Stepanov and Konstantin Serebryany. 2015. MemorySanitizer: Fast detector of uninitialized memory use in C++. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'15)*. IEEE, San Francisco, CA, USA, 46–55. https://doi.org/10.1109/CGO.2015.7054186

[41] Nikos Vasilakis, Achilles Benetopoulos, Shivam Handa, Alizee Schoen, Jiasi Shen, and Martin C. Rinard. 2021. Supply-Chain Vulnerability Elimination via Active Learning and Regeneration. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS'21)*. Association for Computing Machinery, Virtual Event, Republic of Korea, 1755–1770. https://doi.org/10.1145/3460120.3484736

[42] Duc Ly Vu, Ivan Pashchenko, Fabio Massacci, Henrik Plate, and Antonino Sabetta. 2020. Towards Using Source Code Repositories to Identify Software Supply Chain Attacks. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS'20)*. Association for Computing Machinery, Virtual Event, USA, 2093–2095. https://doi.org/10.1145/3372297.3420015

[43] Bowen Wang, Kangjie Lu, Qiushi Wu, and Aditya Pakki. 2021. Unleashing Fuzzing Through Comprehensive, Efficient, and Faithful Exploitable-Bug Exposing. *IEEE Transactions on Dependable and Secure Computing* 19, 5 (May 2021), 2998–3010. https://doi.org/10.1109/TDSC.2021.3079857

[44] Hao Wang, Wenjie Qu, Gilad Katz, Wenyu Zhu, Zeyu Gao, Han Qiu, Jianwei Zhuge, and Chao Zhang. 2022. jTrans: Jump-aware Transformer for Binary Code Similarity Detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'22)*. Association for Computing Machinery, Virtual Event, South Korea, 1–13. https://doi.org/10.1145/3533767.3534367

[45] Shuai Wang and Dinghao Wu. 2017. In-memory Fuzzing for Binary Code Similarity Analysis. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE'17)*. IEEE, Urbana-Champaign, IL, USA, 319–330. https://doi.org/10.1109/ASE.2017.8115645

[46] Xinran Wang, Yoon-Chan Jhi, Sencun Zhu, and Peng Liu. 2009. Behavior Based Software Theft Detection. In *Proceedings of the 16th ACM conference on Computer and communications security (CCS'09)*. ACM Press, Chicago, Illinois, USA, 280–290. https://doi.org/10.1145/1653662.1653696

[47] Xinda Wang, Kun Sun, Archer Batcheller, and Sushil Jajodia. 2019. Detecting "0-Day" Vulnerability: An Empirical Study of Secret Security Patch in OSS. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'19)*. IEEE, Portland, OR, USA, 485–492. https://doi.org/10.1109/DSN.2019.00056

[48] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS'17)*. Association for Computing Machinery, Dallas, Texas, USA, 363–376. https://doi.org/10.1145/3133956.3134018

[49] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. 2017. SemFuzz: Semantics-based Automatic Generation of Proof-of-Concept Exploits. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS'17)*. Association for Computing Machinery, Dallas, Texas, USA, 2139–2154. https://doi.org/10.1145/3133956.3134085

[50] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *Proceedings of the 27th USENIX Security Symposium (SEC'18)*. USENIX Association, Baltimore, MD, 745–761. https://www.usenix.org/conference/usenixsecurity18/presentation/yun

[51] Zheng Zhang, Hang Zhang, Zhiyun Qian, and Billy Lau. 2021. An Investigation of the Android Kernel Patch Ecosystem. In *Proceedings of the 30th USENIX Conference on Security Symposium (SEC'21)*. USENIX Association, Virtual Event, 3649–3666. https://www.usenix.org/conference/usenixsecurity21/presentation/zhang-zheng

[52] Lei Zhao, Yuncong Zhu, Jiang Ming, Yichen Zhang, Haotian Zhang, and Heng Yin. 2020. PatchScope: Memory Object Centric Patch Diffing. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS'20)*. Association for Computing Machinery, Virtual Event, USA, 149–165. https://doi.org/doi.org/10.1145/3372297.3423342

[53] Wenyu Zhu, Zhiyao Feng, Zihan Zhang, Jianjun Chen, Zhijian Ou, Min Yang, and Chao Zhang. 2022. Callee: Recovering Call Graphs for Binaries with Transfer and Contrastive Learning. https://doi.org/10.48550/arXiv.2111.01415 arXiv:2111.01415 [cs].

[54] Fei Zuo, Xiaopeng Li, Patrick Young, Lannan Luo, Qiang Zeng, and Zhexin Zhang. 2019. Neural Machine Translation Inspired Binary Code Similarity Comparison beyond Function Pairs. In *Proceedings 2019 Network and Distributed System Security Symposium (NDSS'19)*. The Internet Society, San Diego, CA, USA, 15. https://www.ndss-symposium.org/ndss-paper/neural-machine-translation-inspired-binary-code-similarity-comparison-beyond-function-pairs/

[55] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2020. ParmeSan: Sanitizer-guided Greybox Fuzzing. In *Proceedings of the 29th USENIX Security Symposium (SEC'20)*. USENIX Association, Virtual Event, 2289–2306. https://www.usenix.org/conference/usenixsecurity20/presentation/osterlund

Received 2023-02-16; accepted 2023-05-03