Facilitating Vulnerability Assessment through PoC Migration

Jiarun Dai*
Fudan University
jrdai14@fudan.edu.cn

Yuan Zhang*
Fudan University
yuanxzhang@fudan.edu.cn

Hailong Xu
Fudan University
18212010076@fudan.edu.cn

Haiming Lyu
Fudan University
20210240108@fudan.edu.cn

Zicheng Wu
Fudan University
20210240111@fudan.edu.cn

Xinyu Xing
Pennsylvania State University
xxing@ist.psu.edu

Min Yang
Fudan University
m_yang@fudan.edu.cn

ABSTRACT

Recent research shows that, even for vulnerability reports archived by MITRE/NIST, they usually contain incomplete information about the software's vulnerable versions, making users of under-reported vulnerable versions at risk. In this work, we address this problem by introducing a fuzzing-based method. Technically, this approach first collects the crashing trace on the reference version of the software. Then, it utilizes the trace to guide the mutation of the PoC input so that the target version could follow the trace similar to the one observed on the reference version. Under the mutated input, we argue that the target version's execution could have a higher chance of triggering the bug and demonstrating the vulnerability's existence. We implement this idea as an automated tool, named VulScope. Using 30 real-world CVEs on 470 versions of software, VulScope is demonstrated to introduce no false positives and only 7.9% false negatives while migrating PoC from one version to another. Besides, we also compare our method with two representative fuzzing tools AFL and AFLGO. We find VulScope outperforms both of these existing techniques while taking the task of PoC migration. Finally, by using VulScope, we identify 330 versions of software that MITRE/NIST fails to report as vulnerable.

CCS CONCEPTS

• **Security and privacy → Software and application security**.

KEYWORDS

Vulnerability Assessment; Trace Alignment; PoC Adjustment

ACM Reference Format:
Jiarun Dai, Yuan Zhang, Hailong Xu, Haiming Lyu, Zicheng Wu, Xinyu Xing, and Min Yang. 2021. Facilitating Vulnerability Assessment through PoC Migration. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21), November 15–19, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3460120.3484594

*co-first authors

CCS '21, November 15–19, 2021, Virtual Event, Republic of Korea

ACM ISBN 978-1-4503-8454-4/21/11…$15.00
https://doi.org/10.1145/3460120.3484594

1 INTRODUCTION

By leveraging the community efforts, MITRE [3] and NIST [4] have indexed more than 150K software vulnerabilities and archived them as the corresponding vulnerability reports. A vulnerability report usually contains vulnerable software versions, the severity of the vulnerability, and even the Proof-of-Concept (PoC) input to reproduce the failure pertaining to the vulnerability. These pieces of information greatly ease patch development, vulnerability management, and security measurement.

However, recent research raises the concern for the quality of vulnerability reports. The works [12, 18, 34, 48] show that incomplete and incorrect vulnerable software versions are prevalent in vulnerability reports. They showcase that incomplete information could leave specific versions of vulnerable software unpatched and expose software end-users to a greater security risk.

To solve this problem, Dong *et al.* [18] replayed PoC inputs – effective for one version – on other versions of the same software. Unfortunately, they discover that, without a further adjustment, a PoC working for one version can barely be migrated for triggering the same bug on other versions. Besides, they also observe that the effort needed for verifying a vulnerability on one version is usually significant. To reduce the manual effort and improve the capability of vulnerability verification, intuition suggests that both *code clone detection* and *patch presence testing* could potentially be applied. In particular, one could quickly determine a vulnerable code's existence with code clone detection [24, 28, 29, 46] and pinpoint a security patch's absence with patch presence testing [16, 25, 52, 55, 57]. Further, he or she could infer whether the version of the software under his or her inspection is potentially vulnerable. However, common static analysis tools could potentially generate high false positives because they cannot confirm the existence of a vulnerability through a PoC input that triggers the target vulnerability and forces the program to accidentally terminate. With a PoC in hand, it can ease patch development, regression testing, and exploitability assessment of a target vulnerability.

In response to these limitations, another instinctive reaction for addressing the problem is to utilize directed fuzzing (e.g., [9, 13]). To be specific, one could first specify the buggy site in the target version

of the software. He or she could then use the reference PoC input as an initial seed for consecutive input mutation. Unlike conventional fuzzing techniques that aim to maximize code coverage (e.g., [1]), directed fuzzing works by steering the fuzzing process towards the designated vulnerable code locations and thus potentially explores paths that can reach out to the specified buggy site. As such, intuition suggests this practice should produce a good outcome in terms of PoC migration. However, as we will show in §5, directed fuzzing does not provide sufficient benefits in generating a new PoC for vulnerability verification. The reason is that directed fuzzing is designed to find a path to the target site quickly but the condition for triggering a target bug is usually encoded only on a couple of critical paths, which are usually ignored by directed fuzzing.

In this work, we propose a PoC migration approach. It adjusts the reference PoC to maximize the similarity between the execution trace generated by the reference PoC on the reference version (called *reference trace*, T_{ref}) and the one generated by the adjusted PoC on the target version (called *target trace*, T_{target}). The key rationale behind our idea is that the execution flows to trigger the same vulnerability on different versions should be similar. Guided by the reference trace, our approach could avoid blindly exploring the huge amount of paths that all head to the buggy sites on the target version, and focuses more on those paths that have already shown to be effective on the reference version. Compared with existing works, our approach has a higher chance and efficiency in generating a new PoC against the vulnerability on the target version.

To realize the idea above, as we will elaborate in §2.1, we tackle two principal challenges. First, we introduce a *cross-version trace alignment* method. Using this method, we correlate the execution traces generated from two different versions of software. Second, we propose a *trace-guided PoC adjustment* approach. With the facilitation of this approach, we correct execution detours, making the target trace approach the reference trace more efficiently. In §3 and §4, we present the details of each of these technical approaches.

To the best of our knowledge, this is the first work that considers migrating PoC across different versions of the same software to perform vulnerability assessment. In practice, confirming a vulnerability is quite difficult, which requires locating buggy code, checking the existence of root cause and verifying the vulnerable conditions for multiple versions of a program. Therefore, it is still challenging for developers (who are familiar with the source code) to confirm a vulnerability. As an automated technique, VulScope can ease the vulnerability assessment for not only developers but also other users who are not familiar with the source code, e.g., testers, security administrators. In comparison with fuzzing techniques for PoC migration, our proposed techniques can complete the task not only effectively but also efficiently. Used in the context of CVE management, our techniques demonstrate significant potential for improving the quality of CVE reports.

In summary, the paper makes the following contributions:

- We propose a systematic approach, named VulScope, to assess vulnerable versions for a target vulnerability by migrating PoC inputs from a reference version to another.
- We design two key techniques to perform the vulnerability assessment: *cross-version trace alignment* and *trace-guided PoC adjustment*.
- We conduct extensive experiments using 30 real-world CVEs on 470 versions of software, and demonstrate VulScope introduces no false positives and only 7.9% false negatives when performing PoC migration. We show VulScope significantly outperforms existing fuzzing techniques in PoC migration tasks.

2 DESIGN OVERVIEW

This section first illustrates our key idea in PoC migration and then gives an overall workflow of our approach. Our detailed approach is presented in §3 and §4.

2.1 Key Idea

The fundamental challenge in our approach is that we need to use the reference trace to guide the migration, but at the same time, we can hardly make the migrated trace (the execution trace generated by the migrated PoC) identical to the reference one due to cross-version code changes. Therefore, we should select an appropriate granularity to leverage the reference trace as guidance, which should balance the benefits brought by reference guidance and the ability to tolerate code changes.

Granularity in Leveraging the Reference Trace. For the purpose of code comparison, several granularities have been used, such as instruction-level [17], basic-block-level [2, 42, 43, 60], function-level [14, 26, 28] and module-level [7, 59]. The granularity to analyze the cross-version traces is quite important to our design. If we guide the migration at a very coarse-grained level, the migration process would explore a lot of code paths that are useless for triggering the target vulnerability. On the contrary, if the guidance is performed at a very fine-grained level, it would limit the exploration ability of the migration process, making it hard to adapt the necessary cross-version code changes.

By examining the unique characteristics of our research problem, we choose function-level as the granularity to leverage the reference trace as guidance. Our design considerations are based on two observations. First, we observe that, in our problem context, the original PoC cannot trigger the same vulnerability on the target version due to cross-version code changes. Compared with instruction-level and basic-block-level, function-level gives better tolerance on the changes across traces which could increase the probability of generating a useful PoC to trigger the same bug on the target version. Second, from the perspective of guidance, we find that module-level is too coarse-grained to provide useful guidance during PoC migration which may dramatically increase the exploration space and reduce the success rate. Therefore, we believe function-level is the most appropriate granularity to guide the PoC migration process with a reference trace.

Challenges. Guided by the reference trace, the PoC migration process can be summarized as two key steps: 1) locates the difference between the reference trace (T_{ref}) and the target trace (T_{target}); 2) adjusts the PoC accordingly to make the generated new T_{target} more similar to the T_{ref}. However, it is non-trivial to follow these steps, due to the various cross-version code changes and the complexity of triggering the same vulnerability. In particular, there are two major challenges in our approach.

Challenge-I: how to align the function calls between cross-version execution traces? Different versions of a program contain various

code changes. Thus, even the same input may generate discrepant execution traces on the reference version and the target version. Actually, the trace discrepancies may be diverse, which include both *syntactic differences* (i.e., behaviors caused by normal code refactoring) and *semantic differences* (i.e., behaviors that are specific to a version or caused by different inputs). To align traces with a lot of differences, existing works mainly require either the same software version [38, 58] (to perform code matching using semantic equivalence) or the same input [27] (to leverage runtime values for code alignment). However, with the aim to generate a useful PoC, we need to modify the original input and then align its execution trace on the target version with the reference trace. Therefore, our approach needs to handle the alignment problem between the execution traces generated with different inputs on different software versions, making existing works hard to apply.

Challenge-II: how to guide the PoC adjustment to trigger the same vulnerability? Based on the alignment between T_{target} and T_{ref}, there may be a lot of trace discrepancies. Though we believe the execution trace to trigger the same vulnerability on different software versions may be quite similar, they inevitably have some different function calls that are specific to a version of a program and cannot be adjusted. Therefore, this kind of trace discrepancy should be identified first and excluded from the scope of PoC adjustment. Besides, for those trace discrepancies that can be adjusted, it is hard to determine which discrepancies need to be adjusted first and how these discrepancies can be efficiently mitigated. In addition, during the PoC adjustment process, we need an accurate vulnerability triage to test whether an input triggers the target vulnerability. These issues together make the whole PoC migration process quite challenging.

Key Techniques. Our basic idea to address the above challenges is to appropriately leverage the reference trace as guidance but also carefully consider the code changes between two versions. Specifically, we propose two techniques to address these challenges.

Technique-I: Cross-version Trace Alignment. As mentioned above, we need to align two traces generated by different inputs on different versions of a program, which may include both syntactic differences and semantic differences. For the syntactic differences, we aim to align them as the same behaviors. For the semantic differences, we should avoid them being aligned. In short, our cross-version trace alignment consists of two steps. First, we perform a function-level code mapping between the two versions of software. Different from existing works, our code alignment not only tolerates function renaming behaviors but also captures function merging/splitting relationships. Second, we propose to use a tree-based structure to represent execution traces and design a tree alignment algorithm for trace alignment. The advantage of tree-based alignment is that it can utilize the context information (e.g., parents nodes) of a function call to guarantee the accuracy of function-level trace alignment (e.g., to avoid aligning functions with semantic differences).

Technique-II: Trace-guided PoC Adjustment. As described in *Challenge-II*, to leverage the reference trace to guide the PoC migration, we need to solve three sub-problems: 1) how to reason the execution detours according to the trace alignment; 2) how to efficiently adjust the inputs to mitigate the execution detours; 3) how to verify the adjusted PoC triggers the same vulnerability?

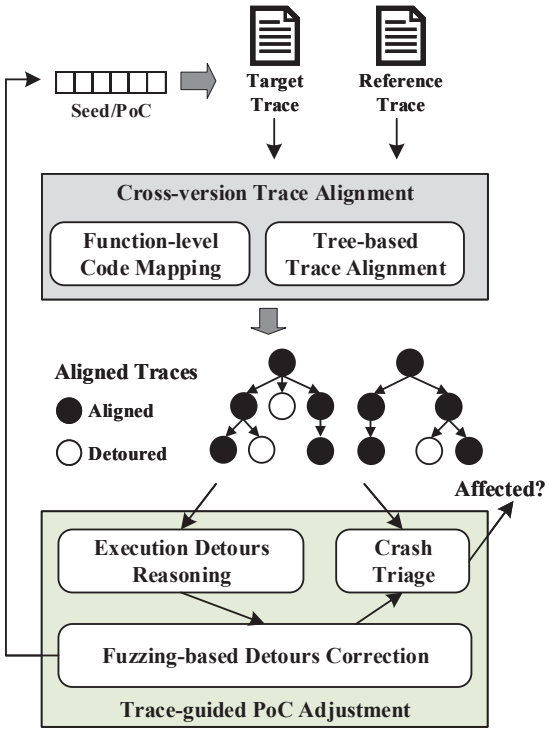

Figure 1: System Overview for PoC Migration.

Accordingly, we address these problems with three new techniques respectively. First, we recognize the execution detours that are specific to a version of software (thus cannot be adjusted) and exclude them from the correction scope. For the remaining detours that could be adjusted, we try to locate the variables that cause these detours in the target trace with program analysis. Second, since the reasoning of which execution detours should be adjusted first and how to accurately adjust them is quite hard, we adopt a fuzzing-based approach here which iteratively mutates the input to keep the T_{target} approaching the T_{ref}, hoping it ultimately trigger the target vulnerability. Third, to verify whether we have triggered the same vulnerability on the target version, we make a conservative design. In particular, we consider crash types, execution trace similarity and buggy function together in determining crashes triggered by the same vulnerability. This design helps our tool to keep a low false positive rate in reporting vulnerable versions.

2.2 Workflow

Following the above key ideas, we design VulScope to facilitate the assessment of vulnerable versions. The overall architecture of VulScope is presented in Figure 1, consisting of two main modules that implement our key techniques respectively. Overall speaking, VulScope takes a PoC that triggers a vulnerability on the reference version as input, and tests whether another version of the same software is also vulnerable to this vulnerability. To confirm the vulnerability, VulScope should generate a new PoC that can indeed trigger the same vulnerability on the specified version. Our core idea is to adopt a trace-aware fuzzing method that continuously adjusts the given PoC, and makes the target version of the software

follow a similar path to that observed on the reference version under the new input. Following this way, intuition suggests that we have a higher chance to generate a new PoC that triggers the same vulnerability on the target version.

In particular, VulScope runs in the following steps:

- *Step-1:* VulScope collects T_{ref} and T_{target} with the given PoC on the reference version and the target version, respectively.
- *Step-2:* VulScope performs *cross-version trace alignment* on T_{ref} and T_{target} to get the aligned functions between the two traces.
- *Step-3:* If a crash is observed on the target version, VulScope uses *crash triage* to verify whether this crash is triggered by the target vulnerability.
- *Step-4:* Based on the aligned cross-version execution traces, VulScope performs *execution detours reasoning* to locate the critical variables that cause these execution detours.
- *Step-5:* VulScope uses *fuzzing-based detours correction* to mutate the input bytes that are related to the critical variables.
- *Step-6:* All the mutated inputs are evaluated with the *crash triage* and given scores based on the similarity between their traces and T_{ref}. If none of the seeds triggers the target vulnerability, they are inserted into a prioritization queue according to their scores. The seed with the highest score will be selected for the next round of mutation (goto *Step-4*).

3 CROSS-VERSION TRACE ALIGNMENT

To correlate functions between two cross-version execution traces, this section first identifies function-level code refactoring changes (i.e., function renaming, function merging/splitting) between two versions of a program (§3.1); then introduces a tree-based data structure to represent the execution trace which naturally provides the running context for every invoked function (§3.2); and finally presents a context-sensitive tree alignment algorithm by incorporating the cross-version function map (§3.3).

3.1 Cross-version Function Mapping

As introduced in §2, we choose to leverage the guidance of the reference trace at the function-level. Thus, a precondition step is to construct a function map between two versions which should be resilient to common code refactoring changes. As summarized in [36], function-level code refactoring changes mainly include function merging/splitting and function renaming. Since functions that share same names between two versions can be directly mapped, we focus on the mapping for those renamed/merged/split functions. In particular, since function-level code similarity [2, 42, 43, 60] is well-studied, we resort to existing works to identify function renaming. Thereafter, we further enhance existing similarity calculation methods to identify function merging/splitting.

Function Renaming Identification. Function renaming is common during version updates to improve code readability. In practice, renamed functions across versions are considered to share similar code semantics with the original ones. To identify "similar" functions with different function names as function renaming pairs, we use existing works in function-level similarity calculation (e.g., [2, 10, 42]). Based on these works, we can set up a "one-to-one function mapping" between two sets of functions by seeking an optimal global matching and identify the matched pairs in the function map as function renaming. The detailed similarity calculation method is presented in Appendix A.

Function Merging/Splitting Identification. As described in [36], function merging/splitting are widely used to avoid code duplication, minimize the number of unnecessary methods, etc. Different from the renamed functions, the relationship between the merged/split functions and the original ones is a kind of "one-to-many function mapping", which is rather complicated. Existing works [22, 49] in recognizing function merging/splitting either require manual workload or complete commit history, thus they are hard to apply in our scenario. To this end, we propose to enhance existing function similarity calculation methods for function merging/splitting identification.

For simplicity, we use how to identify functions that should be merged in the target version to elaborate our approach. First, based on the functions that share same names and those are identified as renamed pairs, we construct a one-to-one function mapping between the reference version and the target version. Second, we construct a call graph for each version and label the matched functions in the graph. Third, we identify the unmatched functions in the target version whose callers/callees have already been matched, named as $S_{unmatched}$. Specifically, for each function X in $S_{unmatched}$, we collect its matched callers and callees, which are named as $S_{candidates}$. We try to merge X with every function Y in $S_{candidates}$ at the corresponding callsite cs, and compute the similarity between the merged function $X + Y$ and the matched pair of Y in the reference version (named Y'). If the similarity score between $X + Y$ and Y' is higher than the similarity score between Y and Y', we identify X and Y should be merged at the callsite cs. At last, we iteratively look for all possible function merging relations in this way. Similarly, the function merging relations in the reference version (aka. function splitting relations in the target version) are identified.

3.2 Tree-structured Execution Trace

Since we need to correlate the execution traces that are generated under different inputs on different versions of software, it is quite challenging to guarantee the accuracy of function alignment. Therefore, we propose to use a tree-based structure to represent the execution trace and perform the trace alignment between two trees. The rationale here is that the context information of a tree node can help to align functions. To be specific, we consider two kinds of context information for a function call:

- *Call Path:* Since a function may have multiple callers, we should keep track of its caller in the context information.
- *Callsite:* Furthermore, since two identical function calls may share the same call path, we also use the callsites (i.e., return address) of a function call in its context.

Definition of Tree-structured Trace. For a program version α, we record all the function calls in an execution with a tree-based data structure $T_\alpha(V_\alpha, E_\alpha)$. In general, $T_\alpha(V_\alpha, E_\alpha)$ is a connected graph which consists of a set of vertices (V_α=$\{v_1, v_2, v_3...\}$) and edges (E_α=$\{e_1, e_2, e_3...\}$), which can be defined as follows:

- *Vertex:* Each vertex $v(f, BBs) \in V_\alpha$ represents a two-tuple function trace entry, consisting of the called function f and the basic-block trace within this function ($BBs = [bb_1, bb_2, bb_3...]$).

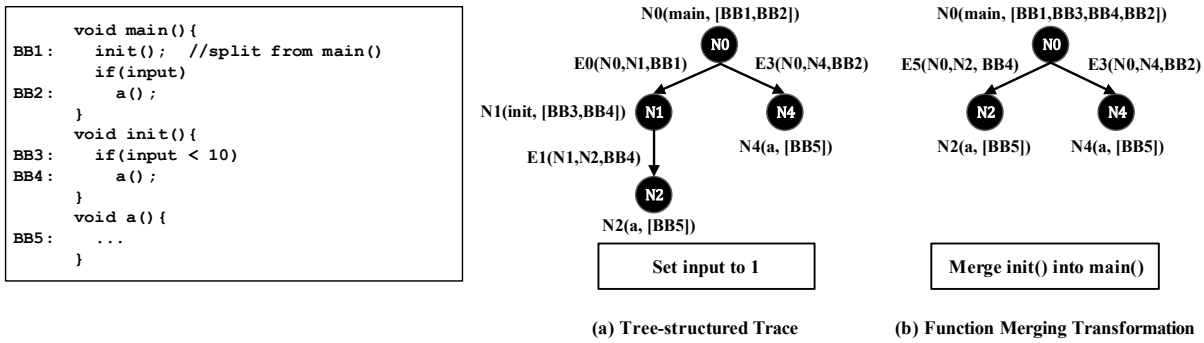

Figure 2: An example to illustrate the tree-structured execution trace and the transformations of function merging/splitting.

A basic block might be executed multiple times (e.g., basic blocks in loops) within a called function.

- *Edge:* Each edge $e(v_i, v_j, cs) \in E_\alpha$ is a three-tuple which refers to a function call, consisting of a vertex $v_i \in V_\alpha$ for the caller function, a vertex $v_j \in V_\alpha$ for the callee function, and the callsite $cs \in v_i.BBs$.

We use the example presented in Figure 2 (a) to illustrate the constructed trace tree when *input* is taken as 1.

Note that T_α is a rooted tree, whose root vertex always calls the *main()* function. Each vertex v except for the root vertex has a single incoming edge and single parent $parent(v)$. If two vertices share the same parent vertex, they are called siblings. Besides, T_α is also an ordered tree, whose siblings are ordered according to the execution. For a given vertex v_i, we can obtain both its caller $parent(v_i)$ and its callsite, named as $callsite(v_i)$, from such trace tree. Therefore, we can perform a context-sensitive trace alignment (see §3.3). Further, its intra-function execution trace $v_i.BBs$ helps to reason fine-grained unaligned parts between two traces (see §4.1).

3.3 Tree-based Trace Alignment

Function Merging/Splitting Transformation. Based on the constructed trace trees, we first try to eliminate the effects of function merging/splitting on the execution traces, so as to ease the cross-version trace alignment. To be specific, we transform the trace tree by merging the functions that are identified as should-be-merged (see §3.1) in both the reference trace and the target trace. We illustrate the detailed procedure as follows:

Suppose that we identify function Y (which is called by X at the callsite cs) should be merged with X on the version α, we first traverse all the edges E_α in the trace tree to locate every edge $e(v_i, v_j, cs) \in E_\alpha$ that satisfies $e.v_i.f = X, e.v_j.f = Y$ and $e.cs = cs$. We take three steps to merge the vertex v_j (split function) into the vertex v_i (caller of split function):

- *Merge intra-function execution trace.* We record the offset o of $e.cs$ in $v_i.BBs$ and then insert $v_j.BBs$ into $v_i.BBs$ at the offset of $o + 1$.
- *Migrate child calls of split function.* We record the index i of v_j among all its siblings and delete v_j from the child vertices of v_i. Then, we migrate all vertices of v_j to be new child vertices of v_i at index i.
- *Update new call relations.* For each migrated vertex of v_j, we add new edges to connect it with v_i.

Taking Figure 2 as an example, function *init()* is identified as a split function from function *main()* during version update. Figure 2 (b) shows the transformed trace tree after merging *init()* into *main()*.

After merging the related function entries in a trace tree, we describe how to align $T_{ref}(V_{ref}, E_{ref})$ to $T_{target}(V_{target}, E_{target})$. First, we introduce the matching criterion for vertices, and then present the alignment algorithm.

Matching Criterion for Vertices. To guarantee the accuracy of aligning function calls across versions, we perform a context-sensitive trace alignment. For the context-sensitiveness, it means we not only check whether a pair of executed functions are identical across versions, but also check whether they share identical running context. Specifically, we set the matching criterion between vertex $v_i \in V_{ref}$ and vertex $v_j \in V_{target}$ as follows:

- *Identical Function:* $v_i.f$ is matched with $v_j.f$.
- *Identical Call Path:* $parent(v_i)$ is matched with $parent(v_j)$.
- *Identical Callsite:* $callsite(v_i)$ is matched with $callsite(v_j)$.

Cross-version Callsite Mapping. As described in the above matching criterion, the trace alignment requires to not only match cross-version functions (see §3.1), but also to match their basic blocks to determine whether two callsites are identical across versions. To be specific, we need to match the basic blocks between $parent(v_i).f$ and $parent(v_j).f$ to determine whether a $callsite(v_i)$ in $parent(v_i).f$ is identical to a $callsite(v_j)$ in $parent(v_j).f$. To this end, we leverage existing basic-block-level similarity calculation methods (e.g., [42, 43, 60]) for basic blocks matching. The detailed similarity calculation method is presented in Appendix A.

Layered Tree Alignment Algorithm. To perform tree alignment, an intuitive approach is to use a tree edit distance algorithm. However, the state-of-art algorithm [41] has a time complexity of $O(n^3)$ that could hardly scale to trees with a large number of vertices. Besides, it does not consider the context information of a vertex during the comparison. Therefore, we design a new tree alignment algorithm that aligns two trees layer by layer. Our approach is feasible for the following two characteristics of our trace tree: ❶ the precondition of two vertices to be matched is that they share identical context (i.e., parent vertices should be matched first); ❷ there would be no cross-layer vertex matching pairs, since we have transformed the tree to eliminate the merged/split functions.

We present the detailed tree alignment procedure as follows: We first directly match the root vertex of $T_{ref}(V_{ref}, E_{ref})$ and

$T_{target}(V_{target}, E_{target})$, which both refer to the *main()* function. Second, for each pair of matched vertices between two trees, we iteratively leverage the longest common subsequence (LCS) algorithm [37, 44] to identify new matched pairs from their child vertices. After the whole alignment procedure, we get the set of unmatched vertices U_{ref} on the reference trace and the set of unmatched vertices U_{target} on the target trace. Thus, $U_{ref} \cup U_{target}$ is viewed as function-level execution detours to be reasoned and corrected during PoC adjustment.

4 TRACE-GUIDED POC ADJUSTMENT

Based on the identified execution detours between the reference trace and the target trace, this section iteratively adjusts the original PoC so as to produce a more similar trace with the reference trace than previous ones, hoping to trigger the same vulnerability in this process. Overall speaking, we have three steps in this phase. First, we pinpoint the critical variables accessed on the target trace (§4.1), which are responsible for the identified execution detours (i.e., changing whose runtime value may help to avoid the detours). Second, intuition suggests that, by concentrating mutation efforts on those input bytes that can influence the pinpointed critical variables, we have a higher chance to correct the execution detours as we desire. In light of this, we propose a fuzzing-based method (§4.2) to iteratively adjust the original PoC input. Third, to determine whether an observed crash during the fuzzing process is triggered by the target vulnerability, we feature a cross-version crash triage mechanism (§4.3).

4.1 Execution Detours Reasoning

As described in §3.3, the execution detours between the reference trace and the target trace can be viewed as the unmatched vertices U_{ref} on the reference trace and the unmatched vertices U_{target} on the target trace. Before reasoning about the execution detours, we remove all child vertices of every unmatched vertex from U_{ref} and U_{target}. Since execution detours may be caused by different reasons, we first classify these detours, and then introduce how to track down the linkage from the execution detours in $U_{ref} \cup U_{target}$ to the corresponding critical variables on the target trace that cause these detours. During the reasoning of execution detours, we would first identify those detours that cannot be corrected (i.e., function calls that cannot happen/avoid in the target trace) and then focus on those detours that can be corrected in the next.

Execution Detours Classification. In general, we find function-level execution detours are caused by 3 reasons.

- *Unexpected Crash* occurs when the target trace triggers another vulnerability and thus crashes before triggering the target vulnerability. To identify an unexpected crash, we rely on cross-version crash triage (§4.3) which determines whether a crash is triggered by the target vulnerability.
- *Missed Call* refers to a function call which is only observed on the reference version while not on the target version. A missed call can be identified as an unmatched vertex v in the reference version ($v \in U_{ref}$) whose parent $parent(v)$ has a matched vertex in the target version.
- *Unintended Call* refers to a function call which is only observed on the target version while not on the reference version. Similarly, an unintended call can be identified as an unmatched vertex v in the target version ($v \in U_{target}$) whose parent $parent(v)$ has a matched vertex in the reference version.

Table 1: Unexpected crash types and the critical variables that need to be mutated for avoiding such crashes.

Crash Type	Critical Variables
Assertion Failure	Checked Variables of Assertion
Floating Point Exception	Denominator Variables
Out-of-bound Access	Index Variables
SEGmentation Violation	Invalid Pointers

Identify Critical Variables. According to the categories of the execution detours, we apply different reasoning strategies to identify the critical variables that are responsible for a detour.

1) Unexpected Crash Reasoning. In this situation, the testcase accidentally triggers another vulnerability which makes the execution crash before triggering the target vulnerability. We identify the critical variables that are responsible for a crash according to its crash type. As summarized in Table 1, we consider four crash types: Assertion Failure, Floating Point Exception (FPE), Out-of-bound Access, and SEGmentation Violation (SEGV). According to this table, the critical variables for each crash type are determined. In the following step, we can mutate the runtime value of the responsible critical variable to avoid such an unexpected crash. Taking FPE as an example, its denominator variable is determined as the critical variable. By mutating the input bytes which may influence the runtime value of it (i.e., making it not equal to 0), we may avoid the testcase from triggering this unexpected crash and thus make the execution proceeds towards the buggy point.

2) Missed Call Reasoning. For a missed call $v_r \in U_{ref}$, its parent vertex $parent(v_r)$ in the reference trace has a matched vertex v_t in the target trace. At first, we determine whether this execution detour can be corrected, by checking the following two conditions: 1) whether $v_r.f$ has a matched function in the target version; 2) whether the matched function of $v_r.f$ in the target version is called by $v_t.f$ with an aligned callsite that has a call from $parent(v_r)$ to v_r in the reference version. If either condition is false, it is impossible to correct this missed call.

Otherwise, we analyze the intra-function execution flow ($v_t.BBs$) of v_t to locate the basic block which detours the execution on the target version. To be specific, we locate a basic block x in $v_t.BBs$ which is the dominator for the basic block that has an aligned callsite to the missed call in the reference version. We then extract the condition variables within the basic block x, which controls the branch taking or not (e.g., the variables used by `CMP` or `SWITCH` instructions), as the critical variables to be adjusted. Figure 3 gives an example. In this example, the target version misses a function call to *_TIFFCheckMalloc()* (line 9). We can determine that line 6 on the target version refers to the basic block which makes the expected callsite (line 9) unreachable. Therefore, the condition variables ($desttypesize$, $direntry{-}{>}tdir\_count$) of line 6 are viewed as critical variables that cause this detour.

3) Unintended Call Reasoning. Similarly, for an unintended call $v_t \in U_{target}$, we first determine whether v_t can be avoided on the target version, by checking whether $v_t.f$ is unconditionally called

```
01 enum TIFFReadDirEntryErr TIFFReadDirEntryArray(
02      tif, direntry, count, desttypesize, value){
03     ......
04     if ((uint64)(2147483647/typesize)<direntry->tdir_count)
05         return(TIFFReadDirEntryErrSizesan);
06     if ((uint64)(2147483647/desttypesize)<direntry->tdir_count)
07         return(TIFFReadDirEntryErrSizesan);
08     ......
09     data=_TIFFCheckMalloc(...)
10 }
```

(b) Intra-function trace of reference PoC on libtiff v4.0.8 (target version)

```
01 enum TIFFReadDirEntryErr TIFFReadDirEntryArrayWithLimit(
02      tif, direntry, count, desttypesize, value, maxcount){
03     ......
04     if ((uint64)(2147483647/typesize)<target_count64)
05         return(TIFFReadDirEntryErrSizesan);
06     if ((uint64)(2147483647/desttypesize)<target_count64)
07         return(TIFFReadDirEntryErrSizesan);
08     ......
09     data=_TIFFCheckMalloc(...)
10 }
```

(a) Intra-function trace of reference PoC on libtiff v4.0.9 (reference version)

Figure 3: An example of reasoning missed calls. Note that the arrows indicate the control flows of the two traces.

by $parent(v_t).f$. If true, such an execution detour could not be corrected.

Otherwise, we perform a backward dominator analysis from the callsite of v_t (aka. $callsite(v_t)$) in the target trace. Through the dominator analysis, we determine the basic block in $parent(v_t).BBs$ whose branching condition controls the reachability of $callsite(v_t)$. We deem the condition variables of this basic block as the critical variables for this unintended call. To illustrate this process, we give an example in Figure 4. The unintended call is the function call *exit()* (line 13) called by *read_samples_pcm()* in Figure 4 (a). Through backward dominator analysis, we first locate line 10 on the target version. While the branch taken at line 10 on the target version cannot help to avoid calling *exit()*. We finally identify its predecessor basic block at line 2 (Figure 4 (a)) which dominates the call of *exit()*. We then determine the condition variable at line 2 ($global.pcmbitwidth$) as the critical variable that causes this detour.

4.2 Fuzzing-based Detours Correction

After associating the identified execution detours with the responsible critical variables in the target trace, we adopt a fuzzing-based method to adjust the original input to mitigate these detours and make the generated execution trace more similar to the reference trace than before.

Fuzzing Loop. Similar to traditional fuzzers, we maintain an input queue to store interesting testcases discovered during the fuzzing process. Initially, we put the reference PoC input into the queue. For each round of fuzzing, we retrieve a seed from the first position of the input queue and collect the execution trace under this seed on the target version. By identifying the execution detours between the collected trace and the reference trace and associating these detours with the critical variables on the target version, we first locate the critical input bytes that affect the runtime values of these critical variables; then mutate these input bytes of the seed to generate a set of new testcases; and finally insert every testcase into the queue

```
01 int read_samples_pcm(musicin, sample_buffer, samples_to_read){
02     switch(global.pcmbitwidth){
03         case 32:
04         case 24:
05         case 16:
06             ......
07         case 8:
08             ......
09         default:
10             if (silent < 10) {
11                 error_printf("...");
12             }
13             exit(1);
14     }
15     ......
16 }
```

(b) Intra-function trace of reference PoC on lame v3.98.4 (target version)

```
01 int read_samples_pcm(musicin, sample_buffer, samples_to_read){
02     switch(global.pcmbitwidth){
03         case 32:
04         case 24:
05         case 16:
06             ......
07         case 8:
08             ......
09         default:
10             if (global_ui_config.silent < 10) {
11                 error_printf("...");
12             }
13             return -1;
14     }
15     ......
16 }
```

(a) Intra-function trace of reference PoC on lame v3.99.5 (reference version)

Figure 4: An example of reasoning unintended calls. Note that the arrows indicate the control flows of the two traces.

according to the similarity between the execution trace (generated by it on the target version) and the reference trace.

During the fuzzing process, if a testcase triggers a crash, we use the crash triage module (see §4.3) to test whether the crash is triggered by the target vulnerability. If so, the fuzzing loop ends and reports this testcase as the desired PoC input for the target version. With a PoC, the target version is verified to be affected by the target vulnerability. Otherwise, the fuzzing process continuously proceeds until a time budget is reached.

The details about the fuzzing loop are described below.

- *Critical Input Bytes Locating.* We first label all critical variables that are identified from execution detours as taint sources. Then, we use dynamic taint analysis to locate the critical input bytes of the seed that affect the runtime values of these critical variables. Technically, we use IR-level instrumentation to implement the variable-level taint tracking (detailed in Appendix A). Although we can pinpoint the input bytes that directly affect the runtime values of the critical variables, we may miss the input bytes that indirectly affect these critical variables, e.g., through control flows. When no input bytes are located via data flows, we also consider the condition variables (i.e., variables that determine which branch to take on conditional jumps) on the execution flow as taint sources, and locate the input bytes that affect these condition variables. All the located input bytes are considered in the mutation stage. We note that this design may introduce undertaint/overtaint issues and we will discuss these issues in §6.

- *Seed Mutation.* For the located critical input bytes of a seed, we apply existing mutation strategies of AFL to mutate their values. In every round of fuzzing, we only mutate a seed, and the time budget for mutating this seed is positively correlated to the number of identified input bytes on this seed.
- *Seed Prioritization.* Within each round of fuzzing, many new seeds are generated after seed mutation. These new seeds are prioritized to insert into the input queue, and the seed with the highest priority will be selected for mutation in the next round of fuzzing. To label a priority for each seed, we use the similarity score between the execution trace collected under this seed on the target version $T_{target}(V_{target}, E_{target})$ and the reference trace $T_{ref}(V_{ref}, E_{ref})$. Specifically, according to the cross-version trace alignment, we first obtain the set of vertices $MATCHED_{ref}$ in the V_{ref} that has matched vertices in the V_{target}, then collect the set of vertices $NOT\_CORRECTABLE_{ref}$ in the V_{ref} who are identified as not-correctable missed calls in execution detours reasoning (see §4.1). We iteratively expand $NOT\_CORRECTABLE_{ref}$ to include all its child vertices. Based on the alignment result, the similarity score between the T_{target} and the T_{ref} is calculated as follows:

$$sim(T_{ref}, T_{target}) = \frac{\left|MATCHED_{ref}\right|}{\left|V_{ref}\right| - \left|NOT\_CORRECTABLE_{ref}\right|} \quad (1)$$

4.3 Crash Triage

When a crash occurs, we need to determine whether the crash is caused by the target vulnerability or not. Though the crash triage mechanism is widely used by fuzzers, it is mainly used for unique crash identification. Existing fuzzers commonly use two heuristics to identify unique crashes: unique stack traces (e.g., used by SYMFUZZ [11]) and unique coverage profile (e.g., used by AFL [1]). However, these heuristics cannot be applied to comparing two crashes that are collected on two different software versions, due to cross-version code changes.

As such, we introduce a tailored triage mechanism for our problem. In our problem, crash triage is used to verify whether a target version is affected by a specified vulnerability. To gain a low false positive rate, we adopt a conservative design in performing such affection assessment. In particular, we set 3 conditions to meet in determining whether two crashes (which are collected on two software versions) are caused by the same vulnerability: ❶ the similarity score between their execution traces (Equation 1) exceeds a pre-defined threshold; ❷ their buggy functions are aligned across versions; and ❸ the bugs they triggered belong to the same category. As we will show in our evaluation (see §5), the above triaging criteria help to keep a low false positive rate in performing vulnerability affection assessment.

5 EVALUATION

This section evaluates the effectiveness and the efficiency of VulScope in migrating a reference PoC to target vulnerable versions of the same software. In particular, it first introduces the experiment setup as well as the data set used for evaluation; then discusses the experiment design; and finally reports the results.

Prototype. We implement a prototype of VulScope, which contains about 4.5K lines of C++ code and 2K lines of Python code (counted by cloc). In our prototype, the static code analysis is implemented on `LLVM 7.0.0` and the fuzzing loop is built upon `AFL 2.56b` [1]. More implementation details are presented in Appendix A.

5.1 Data Set

Methodology. In recent studies [34], researchers have discovered that the vulnerability reports' quality could vary significantly. In order to evaluate our proposed technique, we select real-world vulnerabilities and their reports by following the suggestion provided by one of the studies [34]. To be specific, when selecting our data set, we ensure a report could provide us with the following four information.

- *Availability of Proof-of-Concept input.* We choose only those vulnerability reports containing an external reference link to an available public reference PoC input.
- *Description of vulnerable software versions.* We consider those vulnerability reports that specify the versions of the software vulnerability to that reported security loophole. In this way, we can further assess how well the reports align with the ground truth and the reports of our tool.
- *Guidance of vulnerable software configuration.* For some vulnerabilities, they can be triggered only when the target software is appropriately configured. For example, the target software `libtiff` is vulnerable to the vulnerability associated with `CVE-2018-18557` only if we compile it with the option "`-enable-jbig`". As such, if a report fails to specify the configuration detail of the vulnerable software, we discard the report accordingly.
- *Guidance of triggering method.* For some vulnerabilities, they can be triggered only when a specific running parameter is appropriately specified. For example, the vulnerability associated with `CVE-2015-9101` could be triggered only when "`-f -V 9`" is specified. As a result, we select the report with such details are present.

Data Summary. Following the selection criteria, we randomly selected 30 real-world CVE reports from the National Vulnerability Database (NVD) [4]. These 30 CVE reports cover 6 broadly adopted software in the userspace, covering 6 types of vulnerabilities: heap OOB, stack OOB, divide-by-zero, segmentation fault, integer overflow, and null pointer dereference. To choose the versions of the software for evaluating our tool, we took those versions that never apply the corresponding patches[1] and treated the PoC input associated with the CVE report as our reference PoC input. Table 2 and Appendix-Table 6 summarize the CVEs of our selection, the reference version of the software, the number of target versions we choose for our evaluation, and the security severity of each CVE.

[1]Note that after a vulnerability is identified on a particular version of the software, a software developer usually develops a patch and applies it to a couple of active vulnerable versions. For non-vulnerable versions and inactive-maintained versions, the patches are not applied.

Table 2: Summary of selected vulnerabilities and their corresponding CVE IDs.

Software	CVE	Reference Version	Target Versions[1]	Replay Reference PoC on Target Versions				Affected Versions	
				Crash	Target Crash	Not Target Crash	Not Crash	All	Need PoC Migration[2]
zziplib	CVE-2018-6381	0.13.62	11	11	6	5	0	11	5
	CVE-2017-5976	0.13.62	6	6	6	0	0	6	0
	CVE-2017-5975	0.13.62	6	6	6	0	0	6	0
	CVE-2017-5974	0.13.62	6	6	6	0	0	6	0
audiofile	CVE-2018-17095	0.3.6	7	7	7	0	0	7	0
	CVE-2017-6836	0.3.6	7	6	6	0	1	7	1
	CVE-2017-6834	0.3.6	7	7	7	0	0	7	0
	CVE-2017-6832	0.3.6	7	7	7	0	0	7	0
	CVE-2017-6831	0.3.6	7	7	1	6	0	7	6
	CVE-2017-6835	0.3.6	7	7	7	0	0	7	0
tcpdump	CVE-2017-5485	4.8.1	19	19	19	0	0	19	0
	CVE-2017-13690	4.9.1	21	15	15	0	6	15	0
	CVE-2017-5486	4.8.1	19	19	19	0	0	19	0
	CVE-2017-16808	4.9.2	22	9	9	0	13	9	0
lame	CVE-2017-15046	3.99.5	5	5	5	0	0	5	0
	CVE-2017-15045	3.99.5	9	6	5	1	3	9	4
	CVE-2017-15018	3.99.5	9	9	8	1	0	9	1
	CVE-2015-9101	3.99.5	9	6	5	1	3	9	4
libtiff	CVE-2016-10095	4.0.7	18	13	13	0	5	18	5
	CVE-2016-10269	4.0.7	17	2	0	2	15	12	12
	CVE-2016-10092	4.0.7	17	12	9	3	5	16	7
	CVE-2016-10093	4.0.7	17	11	9	2	6	17	8
	CVE-2018-7456	4.0.9	19	19	19	0	0	19	0
	CVE-2018-12900	4.0.9	19	18	16	2	1	19	3
	CVE-2018-17795	4.0.9	19	0	0	0	19	18	18
	CVE-2018-18557	4.0.9	19	3	3	0	16	19	16
jasper	CVE-2016-9560	1.900.25	21	17	17	0	4	21	4
	CVE-2017-14132	2.0.13	40	40	40	0	0	40	0
	CVE-2018-19540	2.0.14	40	40	40	0	0	40	0
	CVE-2018-19541	2.0.14	40	40	40	0	0	40	0
SUM	30 CVEs	30	470	373	350	23	97	444	94

[1] "Target Versions": Target versions refer to those versions that need vulnerability affection assessment, which are detailed in Appendix-Table 6.
[2] "Need PoC Migration": These versions are vulnerable while the reference PoC inputs fail to trigger the target bugs, thus requiring PoC migration.

As we can observe, the total number of target software versions is 470.

Based on the CVE reports gathered from NVD, among the 470 versions, 103 versions are listed as vulnerable. However, as a recent work [18] has studied, the reported vulnerable versions could be either overestimated or underestimated. As a result, in order to establish the ground truth for our data set (i.e., the ground truth about whether a vulnerability truly exists), we manually examine the vulnerabilities in each version of the software used in our evaluation and listed the number of truly vulnerable versions in Table 2. In total, we manually confirmed 444 versions of software vulnerable to the corresponding vulnerabilities. These vulnerable versions contain 76.8% non-overlapping versions, as specified in the NVD reports. This confirms the overestimating and underestimating discovery by previous research [18].

5.2 Experiment Setup

We run all experiments on an individual virtual machine with 8G memory and 2 CPU cores for each binary. The host of our virtual machine has 24 GB memory and 12 CPU cores (Intel i7-9750H, 2.60 GHz per core). In our experiment, we run our tool (VulScope) for 8 hours while exploring the possibility of PoC migration.

Threshold for Crash Triage ($Threshold_{crash}$). As we mentioned before, our migration techniques use a predefined threshold to perform crash triage. The value selection of $Threshold_{crash}$ is a trade-off between false positives (FP) and false negatives (FN). For example, a low-value choice could cause our tool to mistakenly treat an undesired crash on the target version as a success of PoC migration (i.e., false positives). On the contrary, a higher value could potentially cause our tool to consider a successful migration as an unexpected termination (i.e., false negatives). To ensure VulScope could correctly report the success of the migration, we favor a low FP over a low FN. As such, we choose a relatively high value of 0.7.

5.3 Experiment Design

To evaluate the effectiveness and efficiency of our tool, we design five experiments. Here, we summarize our experiment design below.
Experiment I. We first experiment with the reference PoC input's effectiveness across various target versions (as listed in Appendix-Table 6). Through this experiment, we aim to answer the following questions. For how many vulnerable versions, the reference PoC input could directly generate a crash on the target versions? Among these crashes, how many are the crashes we truly desire, and how

many are unexpected? For those non-crash versions, how many of them require our tool VulScope for PoC migration?

Experiment II. The second experiment evaluates that, for those cases where merely running a PoC input does not generate the desired crash or falsely reports an unexpected termination as a practice of successfully triggering the desired vulnerability, how well our tool VulScope could successfully mutate a reference PoC input and migrate it to vulnerable versions. Besides, we also evaluate whether our proposed technique produces any false positives (i.e., deeming an unsuccessful migration and the corresponding crash as a successful practice). For those versions that VulScope fails to migrate a PoC input, we finally study whether those versions are truly non-vulnerable. If not, we aim to understand the factors that fail VulScope.

Experiment III. We further evaluate the performance of the *cross-version trace alignment*. For vulnerable target versions identified by VulScope in Experiment II, we collect all trace alignment results during PoC migration and crash triage for these versions. With these results, we seek to answer the following questions. What are the sizes of these traces? How common is code refactoring (i.e., function merging/splitting, function renaming) in these traces? How common is the source code change observed in these traces? How effective and efficient is our trace alignment?

Experiment IV. As a fuzzing-based approach optimized to guide a program to a target buggy site, we also design an experiment to compare VulScope with the existing directed fuzzing approach. To the best of our knowledge, the most representative directed fuzzing tools are `AFLGo` [9] and `Hawkeye` [13]. Since `Hawkeye` has not yet been publicly available, we choose `AFLGo`[2] as the baseline for our comparison. In addition to directed fuzzing, we also compare VulScope with the most broadly used fuzzing tool – `AFL`[3]. In this experiment, we select those versions of the software – listed in Table 2 – the reference PoC of which cannot trigger the target bug successfully or triage crashes correctly. As we will discuss later, the total number of versions satisfying this requirement is 94.

It should be noted that both `AFLGO` and `AFL` do not have a clue about whether a crash indicates the success of triggering the desired vulnerability when used for PoC migration. For the directed fuzzing tool `AFLGO`, it even requires manually specifying the buggy site. As such, when performing the comparison mentioned above, we manually mark the target buggy site in the corresponding version of the software. When observing the crash on target versions, we manually examine the crash's root cause, determining whether the crash indicates the success of the desired vulnerability identification and that of PoC migration. According to the time allocated to VulScope, we also set 8 hours for evaluating these tools in migrating PoC inputs. Within the 8 hours, we also record the time that each tool spends on generating the first target crash.

Experiment V. Last but not least, we also design an experiment to assess how well VulScope could be used as a tool to verify a vulnerable version of the software. To achieve this goal, we compare our tool confirmed vulnerable versions with those listed in the reports gathered from NVD. Through this comparison, we aim to understand whether VulScope can be used to facilitate bug report verification, finding those inaccurate vulnerable version claims.

[2]https://github.com/aflgo/aflgo/tree/c2888eb4e6e236549be88d3850831e71d1f0ffa2

[3]https://github.com/mirrorer/afl/tree/2fb5a3482ec27b593c57258baae7089ebdc89043

Table 3: Effectiveness of VulScope in migrating PoC input and performing crash triage.

Ground Truth		VulScope				
P[1]	N[2]	TP	FP	TN	FN	FNR
444	26	409	0	26	35	7.9%

[1] "P" specifies the versions that require our PoC migration and crash triage.
[2] "N" indicates the non-vulnerable versions.

5.4 Experiment Results

Experiment I. Table 2 shows the effect of merely replaying the reference PoC input on various target versions of the same software. As we can observe, 373 out of 470 target versions exhibit crashes and the rest 97 versions do not while taking the corresponding reference PoC input. For each group, we take a closer look and manually analyze the source code of the crashed and non-crashed versions. We discover that, among all the 373 crashing versions, 350 versions successfully trigger the desired bug, and the remaining 23 versions accidentally touch an unexpected bug. For those 97 non-crashed versions, only 26 are no longer vulnerable to the corresponding vulnerabilities. Recall that the goals of our tool are in two-folds. One is to migrate a reference PoC input from one version to another if the vulnerability still exists. The other is to triage the corresponding crash on the target version and correctly distinguish the desired crash from unexpected termination. As such, the number of versions that require our tool's facilitation is 444, including 71 non-crashed vulnerable versions needed for PoC migration and 373 crashes triggered by reference PoC inputs needed for accurate crash triage. In total, this accounts for 94% of our test cases (444/470).

Experiment II. Table 3 summarizes the experiment result, quantifying the effectiveness of our tool in migrating PoC inputs and performing crash triage. As we can observe from the table, VulScope generates 409 crashes and deems them as the consequence of triggering the desired vulnerability. Among these 409 reported successes of PoC migration and crash triage, we note that VulScope introduces no false positives. We further check all the 409 versions that are reported as vulnerable by VulScope. For each CVE, the average amount of time/version between the earliest identified version and the reference version is 48.1 months/13.1 versions.

Besides, Table 3 also shows that VulScope fails to generate a crash indicating the trigger of the target vulnerability for 61 versions. Among these 61 versions, 26 versions are truly unaffected to the corresponding vulnerabilities. It means that the false negatives of VulScope are 35 (FNR: 7.9%=35/444).

To understand the 35 false negatives, we take a closer look, manually analyzing the root cause hidden behind our tool's failure. We discover that failures can be categorized into three practices. ❶ The program is updated to take a new format of user input. Our PoC migration technique cannot correctly mutate an input and thus allow it to pass the newly added format sanity check. We note 9 out of 35 failure cases can be attributed to this practice. ❷ The version change unintentionally adds a sanity check which restricts the trigger of the desired vulnerability through the path observed from the reference version. In order to trigger the vulnerability, one

Table 4: Performance of cross-version trace alignment.

Software	Version Pair (ref, target)	Trace Size	TP	TN	FP	FN	FPR	FNR
zziplib	(0.13.62, 0.13.67)	(40, 36)	70	4	0	2	0%	2.78%
jasper	(1.900.25, 1.900.26)	(9424, 9427)[1]	18116	15	0	720	0%	3.82%
libtiff	(v4.0.7, v3.9.6)	(653, 2676)	612	2595	64	58	2.41%	8.66%
lame	(3.99.5, 3.98.4)	(9738, 9619)[1]	16952	973	0	1432	0%	7.79%
tcpdump	(4.8.1, 4.9.0)	(832, 836)	1664	4	0	0	0%	0%
audiofile	(0.3.5, 0.3.4)	(186, 151)	280	49	0	8	0%	2.78%

[1] We can manually verify such a long trace because over 90% of function calls are in loops.

needs to find an alternative path to reach the buggy site. VulScope is designed to force the target version's execution as similar as that observed on the reference version. As such, it cannot migrate PoC successfully. There are 14 out of 35 failure cases falling into this practice. ❸ The triage module of VulScope also contributes to the failure cases (12 out of 35). We note these failure cases all come from the same software `audiofile`, which result from the significant implementation variation. From version 0.2.7 to version 0.3.6, its developers start to change their implementation from C to C++ gradually. This drastic code change fails our trace-similarity-based comparison in the process of PoC migration.

Experiment III. As presented in Table 3, VulScope identifies 409 vulnerable versions. During PoC migration and crash triage for these 409 vulnerable versions, we perform cross-version trace alignment for 2,919 unique trace pairs. The sizes (number of function calls) of these trace pairs (T_{ref}, T_{target}) vary significantly, ranging from (40, 19) to (20,046, 8,569). In general, there are over 1,634 trace pairs longer than (1000, 1000). From the 2,919 unique trace pairs, VulScope identifies that 85.61% (2,499/2,919) of trace pairs have merged/split functions, while 55.5% (1,619/2,919) of trace pairs involve renamed functions, indicating that function merging/splitting identification is quite important in trace alignment. Besides, for each of the 2,919 unique trace pairs, there are on average 14.6% executed functions which only exist on one version (by comparing function names). For the remaining 85.4% functions which are observed on both versions, 23.8% of them have different codes between two versions. In all, VulScope costs 0.48s to align two traces, and another 13.7s for execution detours reasoning on average.

To evaluate the effectiveness of cross-version trace alignment, we manually investigate 6 trace pairs at different degrees of sizes from different software. Results are shown in Table 4. We find that for most cases, VulScope has no false positives, and the false negative rate is relatively low. We break down these false positives/negatives below.

- *False Positives Breakdown.* We only observe false positives on a trace pair from `libtiff`. During a version update, there are significant code changes in the function *TIFFFetchNormalTag()*, which cause two function calls to *TIFFSetField()* in this function are wrongly aligned. This mistake further makes the child calls of *TIFFSetField()* wrongly aligned.
- *False Negatives Breakdown.* There are two main reasons for the observed false negatives: 1) some renamed functions fail to be mapped due to significant code modifications; 2) code modifications that change the order of two function calls cause the LCS-based tree alignment algorithm hard to align.

In general, though our trace alignment may give some wrong results, it does not significantly affect our PoC adjustment whose fuzzing-based design could to some extent tolerate some inaccuracies in the trace alignment. Besides, we find a large part of FPs/FNs can be further mitigated by introducing more advanced code similarity comparison techniques.

Table 5: Comparison results of AFL, AFLGo, and VulScope.

Time Spent	AFL	AFLGo	VulScope
1h	37	41	71
2h	37	44	71
8h	46	46	71

The detailed migration results of the 3 tools are listed in Appendix-Table 7

Experiment IV. Table 5 presents the performance comparison of AFL, AFLGo, and VulScope (the detailed migration results of the 3 tools are listed in Appendix-Table 7). As we can observe, among all 94 target versions (on which the reference PoC fails to trigger the target bug) used in this experiment, both AFL and AFLGO demonstrate the success of the migration on 46 versions (i.e., 49%). VulScope successfully migrates PoC inputs for 71 versions, which is roughly a 30% increase (i.e., 76% vs 49%). For those versions of software that AFL and AFLGO successfully migrate PoC inputs, we perform further manual analysis and discover that these versions are a subset of those versions that VulScope demonstrates migration success. It means that AFL and AFLGO do not provide additional benefits in terms of PoC migration.

Table 5 also breaks down the success of PoC migration across time. As we can observe, VulScope demonstrates its superiority. In one hour, VulScope could successfully migrate PoC inputs to 71 versions. It is because, in comparison with AFL and AFLGO, VulScope could converge to the expected path (i.e., the path similar to the one observed on the reference version) and thus the buggy site faster.

To further understand the performance of the 3 tools in the task of PoC migration, we investigate how AFL, AFLGO and VulScope behave in approaching the buggy site. Appendix-Table 8 presents the number of seeds that reach the buggy site during the evaluation. Results show 2 situations that AFL and AFLGO fail on the 25 target versions which are only successfully migrated by VulScope.

- *Reach the buggy site but do not trigger the target vulnerability.* In 6 cases, AFL and AFLGO have reached the buggy site for a lot of times, but still fail to trigger the target vulnerability (e.g., CVE-2018-17795). This result clearly demonstrates that (directed) fuzzing is effective in exploring paths (to a buggy site); however, the conditions for triggering a target vulnerability are only encoded on a couple of critical paths among all paths heading to the buggy site. As a comparison, though VulScope does not reach the buggy site for many times, it efficiently triggers the target vulnerability by following the reference trace.
- *Do not reach the buggy site.* In 19 cases, AFL and AFLGO fail to reach the target site (e.g., CVE-2018-18557, CVE-2016-9560, CVE-2017-6831). It is a little surprising that AFLGO has not reached these buggy sites, since it is designed to reach a target site quickly. We find the reason is that AFLGO does not know which bytes to mutate that can help it efficiently approach the target site. For

```
01 int tiffcp(TIFF *in, TIFF *out){
02     uint16 bitspersample, samplesperpixel;
03     uint16 input_compression, input_photometric;
04     ......
05     if(samplesperpixel <= 4)
06         CopyTag(TIFFTAG_TRANSFERFUNCTION, 4, TIFF_SHORT);
07     ......
08 }
```

(a) Intra-function trace of reference PoC on libtiff v4.0.6 (target version)

```
01 int tiffcp(TIFF *in, TIFF *out){
02     uint16 bitspersample, samplesperpixel = 1;
03     uint16 input_compression, input_photometric
04                             = PHOTOMETRIC_MINISBLACK;
05     ......
06     if(samplesperpixel <= 4)
07         CopyTag(TIFFTAG_TRANSFERFUNCTION, 4, TIFF_SHORT);
08     ......
09 }
```

(b) Intra-function trace of reference PoC on libtiff v4.0.7 (reference version)

Figure 5: Case study on CVE-2016-10269. The variable *samplesperpixel* is not initialized on the target version and happens to get a runtime value of 32767 during execution.

example, the original PoC input of CVE-2018-18557 has 6,389 bytes. It is quite inefficient to blindly explore all mutations on all these bytes. By taking the reference trace as guidance, VulScope can efficiently locate the critical input bytes that hinder it from following the reference trace. Therefore, it can not only efficiently reach the buggy site but also trigger the target vulnerability.

Besides, from Appendix-Table 7, we find that VulScope needs more time (about 30 minutes) than AFL and AFLGo to finish PoC migration on several target versions of CVE-2016-10269. We use `libtiff-v4.0.6` (as shown in Figure 5) as a case to investigate the reasons. In this version, a condition variable *samplesperpixel* is not properly initialized and gets a runtime value of 32767. While in the reference version, this variable is initialized to 1. Consequently, it causes a missed call *CopyTag*() on the target version, which prevents the triggering of the target vulnerability. In our experiment, AFL and AFLGo coincidentally modify an input field in the TIFF file which ultimately set *samplesperpixel* to the desired value. As a comparison, VulScope finishes the migration in a similar way but uses more time, because the trace generated by the migrated PoC has a lower similarity with the reference trace.

Experiment V. Appendix-Table 6 summarizes the versions of the software that NVD deems as vulnerable. Besides, the table also lists the number of versions that our tool reports as vulnerable, whereas NVD does not. As we can observe, VulScope could successfully identify 330 vulnerable versions that NVD fails to report. To some extent, this indicates that VulScope can be used as a tool to verify vulnerable versions of software automatically and thus be treated as a potential tool for improving the quality of the NVD reports. It is worth noting that after we reported these missed vulnerable versions to MITRE, they have added all these affected versions to the vulnerability reports.

Besides, we also have a look at the trace similarity between the reference trace and the crashing trace on a newly-identified vulnerable version. We find that although a reference trace differs from a newly-identified crashing trace, the average trace similarity (Equation 1) between them reaches a high value of 96.5%. As evidence, this result demonstrates that the idea of leveraging the reference trace to guide PoC migration is quite practical.

6 DISCUSSION

Using Symbolic Execution to Correct Execution Detours. Correcting execution detours to trigger the target vulnerability is quite difficult, due to the following three facts: 1) some execution detours are not critical in triggering the target vulnerability; 2) our evaluation of cross-version alignment shows that it may report incorrect execution detours; 3) some execution detours may conflict with each other, which means we can only choose part of them to correct. In summary, it is hard to determine an accurate set of detours that should be corrected. Therefore, we cannot apply symbolic execution here because it requires to know exactly which detours to correct, not to mention that its constraint solving is heavy-weight and may get stuck on complicated constraints. In contrast, our fuzzing-based approach iteratively corrects these detours without knowing which part of detours should be corrected, only if the correction process keeps the global similarity with the reference trace improving.

Applications to Closed-source Binaries. The high-level idea in PoC migration (i.e., collecting a reference trace as guidance to adjust the input on the target version) is applicable to closed-source binaries. However, our current implementation relies on some techniques that require source code: trace collection, taint analysis and code similarity comparison. For trace collection and taint analysis on binaries, we can use binary-level instrumentation tools (e.g., Pin tools [5]). For binary-level code similarity comparison, there are also plenty of techniques [19, 32] that can be used which support cross-version, cross-optimization, and obfuscation-resilient code matching on both function-level and basic-block-level.

Limitations and Future Work. Though VulScope has outperformed both AFL and AFLGo in terms of completing the task of PoC migration, it still has several limitations.

- VulScope adopts backward taint analysis to locate the critical inputs that may influence the runtime values of the critical variables. Therefore, it may meet overtaint and undertaint issues. In particular, the overtainting may hurt its efficiency by wasting mutation efforts on the input bytes that are not useful for execution detours correction; the undertainting may affect its effectiveness by missing some critical input bytes. Nevertheless, these issues do not introduce false positives/negatives in our evaluation. A potential mitigation strategy here is to use input probing techniques [21, 56].
- Our tree alignment algorithm adopts a layer-by-layer design, which may still wrongly align function calls in the face of large code changes. Besides, this design may lead to propagating the wrong mappings from a layer to its following layers. In our evaluation, the trace alignment module reports some incorrect results, which are caused by the imprecise matching between functions or basic blocks. Although our fuzzing-based PoC adjustment can to some extent tolerate such alignment errors, we can improve the trace alignment by incorporating more advanced techniques in code similarity comparison [19, 32].
- VulScope needs to tolerate some common code refactoring across different program versions, such as function renaming

and function merging/splitting. However, the proposed technique might fail to match functions with complicated refactoring occurs (e.g., a function is merged and split at the same time). We plan to investigate these problems in the future.

- We consider four common types of unexpected crashes when reasoning execution detours. When other types of unexpected crashes occur, we cannot associate them with some critical variables. Instead, we resort to reasoning the control-flow execution detours (i.e., missed calls and unintended calls) observed on the crashing paths to the reference trace. We plan to support more crash types by leveraging more heavy-weight crash analysis [8, 53, 54].
- We need to traverse a call graph when identifying function merging/splitting pairs. An implementation issue here is that we do not consider indirect calls in the call graph construction, so it may introduce false negatives. We plan to improve our prototype by leveraging an indirect call target analysis [33].
- Our key idea for PoC migration is that following a similar path to the reference trace may help to trigger the same vulnerability on another version. This observation helps VulScope to achieve quite good performance in migrating a PoC input to another version of real-world programs. However, there might be situations where the vulnerability-triggering conditions encoded on the reference trace are no longer satisfied on the target version. An extreme case that we observe in the evaluation is that the target version even does not support the parsing of reference PoC (aka unsupported file format), though it is still vulnerable to the same vulnerability. To handle these cases, VulScope can be enhanced to increase its vulnerability exploration capability, perhaps with the guidance of root cause analysis [8, 35].

7 RELATED WORK

Trace Alignment is used to correlate two execution traces. Zhang *et al.* [58] and Nagarajan *et al.* [38] aim to align traces collected from semantic-identical targets for the purpose of software piracy detection, debugging, etc. Unfortunately, these methods are not suitable in our problem context, since PoC migration requires aligning traces generated from different versions of a program. To take a step further, Kargen *et al.* [27] study the problem of aligning traces between semantic similar (not identical) targets. However, their approach is built upon runtime-value-based analysis, which requires both traces to be collected under the same input. Therefore, this method is also inapplicable in our work, because our traces are generated under different inputs. Hoffman *et al.* [23] present a trace alignment technique on traces collected from two versions of a program. Similar to our work, they also try to identify code refactoring changes to help the trace alignment. Different from our work, their method requires collecting a lot of traces under the same legal inputs on two versions of a program and then identifies those commonly-observed trace differences as code refactoring. Apparently, such a method is limited in its coverage. Different from all existing works, our work handles the alignment problem between cross-version execution traces generated with different inputs.

Code Similarity is widely-used to identify the cloned code snippets in a target software. Code similarity calculation can be applied at different granularities, including instruction-level [17], basic-block-level [2, 42, 43, 60], function-level [14, 26, 28], module-level [7, 59], etc. In our work, we resort to existing function-level and basic-block-level similarity calculation methods to correlate cross-version functions and corresponding callsites. To compare similarity, these works commonly extract internal and structural features of code elements: internal features capture the characteristic of the code element itself (e.g., opcodes, constant data access), while structural features represent relations between code elements (e.g., caller-callee relations, predecessor-successor relations).

Directed Fuzzing is a specific kind of fuzzing approach. Different from conventional fuzzing that aims to explore maximum code coverage, directed fuzzing is designed to generate a series of concrete inputs that could direct the target program to reach out to the desired code fragment (e.g., `AFLGo` [9] and `Hawkeye` [13]). To fulfill the goal, prior research works have introduced various methods to guide input mutation (e.g., adjusting input with the guidance of target sequence [30, 31, 39], sanitizer checks [15, 40], memory usage [51], and even typestate [50]). As is mentioned and compared in the previous sections, directed fuzzing can also potentially be used for solving our problem. However, directed fuzzing suffers from finding the critical path to trigger the desired bug even though it could find paths to reach the desired code snippet. As such, it cannot be treated as an effective solution for our problem.

8 CONCLUSION

Crowd-sourced software vulnerabilities are usually reported as CVE reports and later achieved by the National Vulnerability Database (NVD). A recent study has already unveiled these reports' low-quality issues (e.g., over-claimed or under-claimed vulnerable versions). In this work, we introduce a systematic, automated approach to assessing the under-claimed vulnerable versions for a reported vulnerability. Technically, our proposed approach utilizes a fuzzing-based method to migrate a Proof-of-Concept input to a target version. We show that the fuzzing-based approach could be an effective, efficient approach to migrating a PoC input from one vulnerable version to another through a series of carefully designed experiments. We conclude that our proposed technique can serve as a tool to facilitate the improvement of NVD's report quality. As part of our future work, we will conduct a larger scale of experiments further exploring the utility of our tool.

ACKNOWLEDGMENT

We would like to thank our shepherd Yajin Zhou and the anonymous reviewers for their insightful comments that helped improve the paper. This work was supported in part by National Natural Science Foundation of China (U1836210, U1836213, U1736208, 61972099, 62172105), and Natural Science Foundation of Shanghai (19ZR1404800). Yuan Zhang was supported in part by the Shanghai Rising-Star Program under Grant 21QA1400700. Min Yang is the corresponding author, and a faculty of Shanghai Institute of Intelligent Electronics & Systems, Shanghai Institute for Advanced Communication and Data Science, and Engineering Research Center of Cyber Security Auditing and Monitoring, Ministry of Education, China.

REFERENCES

[1] 2021. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/.
[2] 2021. BinDiff. https://www.zynamics.com/bindiff.html.
[3] 2021. Common Vulnerabilities and Exposures. https://cve.mitre.org/.
[4] 2021. National Vulnerability Database. https://nvd.nist.gov/.
[5] 2021. PIN Tools. https://software.intel.com/content/www/us/en/develop/articles/pin-a-dynamic-binary-instrumentation-tool.html.
[6] 2021. PolyTracker. https://github.com/trailofbits/polytracker.
[7] Michael Backes, Sven Bugiel, and Erik Derr. 2016. Reliable Third-party Library Detection in Android and its Security Applications. In *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS), Vienna, Austria*. 356–367.
[8] Tim Blazytko, Moritz Schlögel, Cornelius Aschermann, Ali Abbasi, Joel Frank, Simon Wörner, and Thorsten Holz. 2020. AURORA: Statistical Crash Analysis for Automated Root Cause Explanation. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security), Virtual Event, USA*. 235–252.
[9] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS), Dallas, TX, USA*. 2329–2344.
[10] Martial Bourquin, Andy King, and Edward Robbins. 2013. Binslayer: Accurate Comparison of Binary Executables. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop (PPREW), Rome, Italy*. 4:1–4:10.
[11] Sang Kil Cha, Maverick Woo, and David Brumley. 2015. Program-adaptive Mutational Fuzzing. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P), San Jose, CA, USA*. 725–741.
[12] Oscar Chaparro, Jing Lu, Fiorella Zampetti, Laura Moreno, Massimiliano Di Penta, Andrian Marcus, Gabriele Bavota, and Vincent Ng. 2017. Detecting Missing Information in Bug Descriptions. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (FSE), Paderborn, Germany*. 396–407.
[13] 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 25th ACM SIGSAC Conference on Computer and Communications Security (CCS), Toronto, ON, Canada*. 2095–2108.
[14] Kai Chen, Peng Liu, and Yingjun Zhang. 2014. Achieving Accuracy and Scalability Simultaneously in Detecting Application Clones on Android Markets. In *Proceedings of the 36th International Conference on Software Engineering (ICSE), Hyderabad, India*. 175–186.
[15] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. 2020. SAVIOR: Towards Bug-Driven Hybrid Testing. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P), San Francisco, CA, USA*. 1580–1596.
[16] Jiarun Dai, Yuan Zhang, Zheyue Jiang, Yingtian Zhou, Junyan Chen, Xinyu Xing, Xiaohan Zhang, Xin Tan, Min Yang, and Zhemin Yang. 2020. BScout: Direct Whole Patch Presence Test for Java Executables. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security), Virtual Event, USA*. 1147–1164.
[17] 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), Santa Barbara, CA, USA*. 266–280.
[18] Ying Dong, Wenbo Guo, Yueqi Chen, Xinyu Xing, Yuqing Zhang, and Gang Wang. 2019. Towards the Detection of Inconsistencies in Public Security Vulnerability Reports. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security), Santa Clara, CA, USA*. 869–885.
[19] Yue Duan, Xuezixiang Li, Jinghan Wang, and Heng Yin. 2020. DeepBinDiff: Learning Program-Wide Code Representations for Binary Diffing. In *Proceedings of the 27th Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA*.
[20] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. 2016. discovRE: Efficient Cross-architecture Identification of Bugs in Binary Code. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA*.
[21] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. 2020. GREYONE: Data Flow Sensitive Fuzzing. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security), Virtual Event, USA*. 2577–2594.
[22] Michael W. Godfrey and Lijie Zou. 2015. Using Origin Analysis to Detect Merging and Splitting of Source Code Entities. In *Proceedings of the IEEE Trans. Software Eng. (TSE)*. 166–181.
[23] Kevin J. Hoffman, Patrick Eugster, and Suresh Jagannathan. 2009. Semantics-aware Trace Analysis. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Dublin, Ireland*. 453–464.
[24] Jiyong Jang, Abeer Agrawal, and David Brumley. 2012. ReDeBug: Finding Unpatched Code Clones in Entire OS Distributions. In *Proceedings of the 23rd IEEE Symposium on Security and Privacy (S&P), San Francisco, CA, USA*. 48–62.
[25] Zheyue Jiang, Yuan Zhang, Jun Xu, Qi Wen, Zhenghe Wang, Xiaohan Zhang, Xinyu Xing, Min Yang, and Zhemin Yang. 2020. PDiff: Semantic-based Patch Presence Testing for Downstream Kernels. In *Proceedings of the 27th ACM SIGSAC Conference on Computer and Communications Security (CCS), Virtual Event, USA*. 1149–1163.
[26] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. In *Proceedings of the IEEE Trans. Software Eng. (TSE)*. 654–670.
[27] Ulf Kargén and Nahid Shahmehri. 2017. Towards Robust Instruction-level Trace Alignment of Binary Code. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), Urbana, IL, USA*. 342–352.
[28] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. 2017. VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (S&P), San Jose, CA, USA*. 595–614.
[29] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. VulDeePecker: A Deep Learning-based System for Vulnerability Detection. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA,*.
[30] Hongliang Liang, Lin Jiang, Lu Ai, and Jinyi Wei. 2020. Sequence Directed Hybrid Fuzzing. In *Proceedings of the 27th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), London, ON, Canada*. 127–137.
[31] Hongliang Liang, Yini Zhang, Yue Yu, Zhuosi Xie, and Lin Jiang. 2019. Sequence Coverage Directed Greybox Fuzzing. In *Proceedings of the 27th International Conference on Program Comprehension (ICPC), Montreal, QC, Canada*. 249–259.
[32] 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), Montpellier, France*. 667–678.
[33] Kangjie Lu and Hong Hu. 2019. Where Does It Go? Refining Indirect-call Targets with Multi-layer Type Analysis. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS), London, UK*. 1867–1881.
[34] Dongliang Mu, Alejandro Cuevas, Limin Yang, Hang Hu, Xinyu Xing, Bing Mao, and Gang Wang. 2018. Understanding the Reproducibility of Crowd-reported Security Vulnerabilities. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security), Baltimore, MD, USA*. 919–936.
[35] Dongliang Mu, Wenbo Guo, Alejandro Cuevas, Yueqi Chen, Jinxuan Gai, Xinyu Xing, Bing Mao, and Chengyu Song. 2019. RENN: Efficient Reverse Execution with Neural-network-assisted Alias Analysis. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), San Diego, CA, USA*. 924–935.
[36] Emerson R. Murphy-Hill, Chris Parnin, and Andrew P. Black. 2012. How We Refactor, and How We Know It. In *Proceedings of the IEEE Trans. Software Eng. (TSE)*. 5–18.
[37] Eugene W. Myers. 1986. An O(ND) Difference Algorithm and its Variations. In *Algorithmica*. 251–266.
[38] Vijayanand Nagarajan, Rajiv Gupta, Matias Madou, Xiangyu Zhang, and Bjorn De Sutter. 2007. Matching Control Flow of Program Versions. In *Proceedings of the 23rd IEEE International Conference on Software Maintenance (ICSM), Paris, France*. 84–93.
[39] Manh-Dung Nguyen, Sébastien Bardin, Richard Bonichon, Roland Groz, and Matthieu Lemerre. 2020. Binary-level Directed Fuzzing for Use-After-Free Vulnerabilities. In *Proceedings of the 23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID), Virtual Event, USA*. 47–62.
[40] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Giuffrida Cristiano. 2020. ParmeSan: Sanitizer-guided Greybox Fuzzing. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security), Virtual Event, USA*. 2289–2306.
[41] Mateusz Pawlik and Nikolaus Augsten. 2016. Tree Edit Distance: Robust and Memory-efficient. In *Proceedings of the Information Systems (Inf. Syst.)*. 157–173.
[42] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. 2015. Cross-architecture Bug Search in Binary Executables. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P), San Jose, CA, USA*. 709–724.
[43] Jannik Pewny, Felix Schuster, Lukas Bernhard, Thorsten Holz, and Christian Rossow. 2014. Leveraging Semantic Signatures for Bug Search in Binary Programs. In *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC), New Orleans, LA, USA*. 406–415.
[44] John W. Ratcliff and David E. Metzener. 1998. Ratcliff-obershelp Pattern Recognition. In *Dictionary of Algorithms and Data Structures (DADS)*.
[45] Eric Sven Ristad and Peter N Yianilos. 1998. Learning String-edit Distance. In *Proceedings of the IEEE Trans. Pattern Anal. Mach. Intell. (TPAMI)*. 522–532.
[46] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. 2016. SourcererCC: Scaling Code Clone Detection to Big-code. In *Proceedings of the 38th International Conference on Software Engineering (ICSE), Austin, TX, USA*. 1157–1168.
[47] Pang-Ning Tan et al. 2006. *Introduction to Data Mining*.
[48] Xin Tan, Yuan Zhang, Chenyuan Mi, Jiajun Cao, Kun Sun, Yifan Lin, and Min Yang. 2021. Locating the Security Patches for Disclosed OSS Vulnerabilities with Vulnerability-Commit Correlation Ranking. In *Proceedings of the 28th ACM SIGSAC Conference on Computer and Communications Security (CCS), Virtual Event, Republic of Korea*.

[49] Nikolaos Tsantalis, Matin Mansouri, Laleh Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and Efficient Refactoring Detection in Commit History. In *Proceedings of the 40th International Conference on Software Engineering (ICSE), Gothenburg, Sweden.* 483–494.
[50] Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, and Yulei Sui. 2020. Typestate-Guided Fuzzer for Discovering Use-after-Free Vulnerabilities. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE), Seoul, South Korea.* 999–1010.
[51] Cheng Wen, Haijun Wang, Yuekang Li, Shengchao Qin, Yang Liu, Zhiwu Xu, Hongxu Chen, Xiaofei Xie, Geguang Pu, and Ting Liu. 2020. MemLock: Memory Usage Guided Fuzzing. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE), Seoul, South Korea.* 765–777.
[52] Yang Xiao, Bihuan Chen, Chendong Yu, Zhengzi Xu, Zimu Yuan, Feng Li, Binghong Liu, Yang Liu, Wei Huo, Wei Zou, and Wenchang Shi. 2020. MVP: Detecting Vulnerabilities using Patch-enhanced Vulnerability Signatures. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security), Virtual Event, USA.* 1165–1182.
[53] Jun Xu, Dongliang Mu, Ping Chen, Xinyu Xing, Pei Wang, and Peng Liu. 2016. Credal: Towards Locating A Memory Corruption Vulnerability with Your Core Dump. In *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS), Vienna, Austria.* 529–540.
[54] Jun Xu, Dongliang Mu, Xinyu Xing, Peng Liu, Ping Chen, and Bing Mao. 2017. Postmortem Program Analysis with Hardware-enhanced Post-crash Artifacts. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security), Vancouver, BC, Canada.* 17–32.
[55] Yifei Xu, Zhengzi Xu, Bihuan Chen, Fu Song, Yang Liu, and Ting Liu. 2020. Patch Based Vulnerability Matching for Binary Programs. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), Virtual Event, USA.* 376–387.
[56] Wei You, Xueqiang Wang, Shiqing Ma, Jianjun Huang, Xiangyu Zhang, XiaoFeng Wang, and Bin Liang. 2019. ProFuzzer: On-the-fly Input Type Probing for Better Zero-Day Vulnerability Discovery. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P), San Francisco, CA, USA.* 769–786.
[57] Hang Zhang and Zhiyun Qian. 2018. Precise and Accurate Patch Presence Test for Binaries. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security), Baltimore, MD, USA.* 887–902.
[58] Xiangyu Zhang and Rajiv Gupta. 2005. Matching Execution Histories of Program Versions. In *Proceedings of the 10th European Software Engineering Conference (ESEC), Lisbon, Portugal.* 197–206.
[59] Yuan Zhang, Jiarun Dai, Xiaohan Zhang, Sirong Huang, Zhemin Yang, Min Yang, and Hao Chen. 2018. Detecting Third-party Libraries in Android Applications with High Precision and Recall. In *IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), Campobasso, Italy.* 141–152.
[60] 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 of the 26th Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA.*

A IMPLEMENTATION DETAILS

Trace Collection. We implement a lightweight trace collector using an `LLVM` instrumentation pass. For every function call, we log the callee function, caller function, and the corresponding callsite. These logs are used to construct tree-structured execution trace. We also log the intra-function execution flow by instrumenting each basic block. Each basic block has a unique intra-function ID.

Code Similarity Calculation. We leverage code features that are widely used in existing works to calculate similarity scores between cross-version functions and basic blocks.

For function-level similarity calculation, we first normalize the source code of a function, and then extract various features from the function, including its callers, callees and constant values. We define $Sim(X, Y)$ as the similarity score between function `X` and function `Y`. When performing cross-version function matching, only those function pairs whose similarity score exceeds a predefined threshold are considered as matched pairs. As suggested by [46], this threshold is set to 0.7 in our prototype. Specifically, $Sim(X, Y)$ is calculated as follows.

$$Sim(X,Y) = \frac{Sim_c(X,Y) + Sim_j(X,Y)}{2} \tag{2}$$

$$Sim_c(X,Y) = 1 - \frac{edit\_distance(X,Y)}{max(len(X), len(Y))} \tag{3}$$

$$Sim_j(X,Y) = \frac{Sim_{caller}(X,Y) + Sim_{callee}(X,Y) + Sim_{const}(X,Y)}{3} \tag{4}$$

$$Sim_{caller}(X,Y) = Jaccard(F_{caller}(X), F_{caller}(Y)) \tag{5}$$

$$Sim_{callee}(X,Y) = Jaccard(F_{callee}(X), F_{callee}(Y)) \tag{6}$$

$$Sim_{const}(X,Y) = Jaccard(F_{const}(X), F_{const}(Y)) \tag{7}$$

In the above equations, $edit\_distance()$ calculates the edit distance [45] between the normalized source code of the functions; $len()$ indicates the length of the normalized function code. The $F_{caller}()$, $F_{callee}()$ and $F_{const}()$ extract the callers of the function, the callees of the function, and the constant values used in the function respectively. $Jaccard()$ takes two feature sets as input and returns the Jaccard similarity [47] between them.

For basic-block-level similarity calculation, we consider the internal features and structural features of a basic block. Given a pair of basic blocks (b_r, b_t), we use Sim_b to represent their similarity score on internal features and Sim_s to represent their similarity score on structural features. To compute Sim_b, we collect all the internal features in the basic block, including the opcodes, the data, and the called functions, and use Jaccard similarity to measure how many features are shared between two basic blocks, i.e., $Sim_b(b_r, b_t) = Jaccard(b_r, b_t)$. To compute Sim_s, we first collect the predecessor basic blocks of b_r and b_t as $\{P_r\}$ and $\{P_t\}$ and their successor basic blocks as $\{Q_r\}$ and $\{Q_t\}$. Then, we define Sim_s to measure the similarity of their predecessors as well as that of their successor with the equation below.

$$Sim_s(\{X_r\}, \{X_t\}) = \frac{max(\sum_{x_t \in \{X_t\}}^{x_r \in \{X_r\}} Sim_b(x_r, x_t))}{min(|\{X_r\}|, |\{X_t\}|)} \tag{8}$$

Finally, we combine the similarity score of the internal features and the similarity score of the context features to calculate the similarity score between b_r and b_t.

$$Sim(b_r, b_t) = \frac{Sim_b(b_r, b_t) + Sim_s(\{P_r\}, \{P_t\}) + sim_s(\{Q_r\}, \{Q_t\})}{3} \tag{9}$$

By measuring the similarity score, a pair of basic blocks (b_r, b_t) can be viewed as a candidate pair for alignment only when their similarity score exceeds a pre-defined threshold. In this work, we set the threshold to 0.5 which is the same with the threshold in [20].

Taint Tracking. We correlate the input bytes with the critical variables using variable-level taint tracking. In our implementation, this component is built on the top of `PolyTracker` [6]. Since the official version of `PolyTracker` provides only a function-to-input-bytes mapping, we enhance its tainting logic on load and store instructions to make it support fine-grained variable-to-input-bytes mapping. Besides, our implementation also optimizes `Polytracker` to support some language features that are used in real-world programs. For example, *variable-length* argument is not well-supported in the original version of `PolyTracker`. This could result in under-tainted variables at the runtime. To solve this problem, we first use static analysis to determine the concrete length of encountered variable-length argument at the function callsite; then store the taint labels of all these arguments. Further, we instrument the *va_arg* macro for propagating the taint labels of arguments when they are used.

Table 6: The details about the 30 CVEs to evaluate VulScope.

Software	CVE	Vulnerability Type	Range of Unpatched Versions	Range of Release Time	CVSS 3.x Score	Vulnerable Version (NVD)	Newly Detected Versions by VulScope
zziplib	CVE-2018-6381	Heap Overflow	[0.13.56, 0.13.67] (12)	[2009.6, 2017.6]	6.5 MEDIUM	0.13.67	11
	CVE-2017-5976	Heap Overflow	[0.13.56, 0.13.62] (7)	[2009.6, 2012.3]	5.5 MEDIUM	0.13.62	6
	CVE-2017-5975	Heap Overflow	[0.13.56, 0.13.62] (7)	[2009.6, 2012.3]	5.5 MEDIUM	0.13.62	6
	CVE-2017-5974	Segmentation Fault	[0.13.56, 0.13.62] (7)	[2009.6, 2012.3]	5.5 MEDIUM	0.13.62	6
audiofile	CVE-2018-17095	Heap Overflow	[0.2.7, 0.3.6] (8)	[2010.3, 2013.3]	8.8 HIGH	0.3.6	6
	CVE-2017-6836	Integer Overflow to OOB	[0.2.7, 0.3.6] (8)	[2010.3, 2013.3]	5.5 MEDIUM	0.3.6	6
	CVE-2017-6834	Integer Overflow to OOB	[0.2.7, 0.3.6] (8)	[2010.3, 2013.3]	5.5 MEDIUM	0.3.6	7
	CVE-2017-6832	Heap Overflow	[0.2.7, 0.3.6] (8)	[2010.3, 2013.3]	5.5 MEDIUM	0.3.6	7
	CVE-2017-6831	Heap Overflow	[0.2.7, 0.3.6] (8)	[2010.3, 2013.3]	5.5 MEDIUM	0.3.6	7
	CVE-2017-6835	Divided By Zero	[0.2.7, 0.3.6] (8)	[2010.3, 2013.3]	5.5 MEDIUM	0.3.6	0
tcpdump	CVE-2017-5485	Heap Overflow	[3.9.1, 4.8.1] (20)	[2005.10, 2016.10]	9.8 CRITICAL	< 4.9.0	0
	CVE-2017-13690	Heap Overflow	[3.9.1, 4.9.1] (22)	[2005.10, 2017.7]	9.8 CRITICAL	< 4.9.2	0
	CVE-2017-5486	Heap Overflow	[3.9.1, 4.8.1] (20)	[2005.10, 2016.10]	9.8 CRITICAL	< 4.9.0	0
	CVE-2017-16808	Heap Overflow	[3.9.1, 4.9.2] (23)	[2005.10, 2017.9]	5.5 MEDIUM	< 4.9.3	0
lame	CVE-2017-15046	Stack Overflow	[3.97, 3.98.4], [3.99.4, 3.99.5] (6)[1]	[2006.9, 2010.3], [2012.1, 2012.2]	5.5 MEDIUM	3.99.5	5
	CVE-2017-15045	Heap Overflow	[3.97, 3.99.5] (10)	[2006.9, 2012.2]	5.5 MEDIUM	3.99.5	8
	CVE-2017-15018	Heap Overflow	[3.97, 3.99.5] (10)	[2006.9, 2012.2]	5.5 MEDIUM	3.99.5	8
	CVE-2015-9101	Heap Overflow	[3.97, 3.99.5] (10)	[2006.9, 2012.2]	5.5 MEDIUM	3.99.5	8
libtiff	CVE-2016-10095	Stack Overflow	[3.9.3, 4.0.8] (19)	[2010.6, 2017.5]	5.5 MEDIUM	4.0.7	13
	CVE-2016-10269	Heap Overflow	[3.9.3, 4.0.7] (18)	[2010.6, 2016.11]	7.8 HIGH	4.0.7	12
	CVE-2016-10092	Heap Overflow	[3.9.3, 4.0.7] (18)	[2010.6, 2016.11]	7.8 HIGH	4.0.7	16
	CVE-2016-10093	Integer Overflow to OOB	[3.9.3, 4.0.7] (18)	[2010.6, 2016.11]	7.8 HIGH	4.0.7	17
	CVE-2018-7456	Null Pointer Dereference	[3.9.3, 4.0.9] (20)	[2010.6, 2017.11]	6.5 MEDIUM	4.0.9	19
	CVE-2018-12900	Integer Overflow to OOB	[3.9.3, 4.0.9] (20)	[2010.6, 2017.11]	8.8 HIGH	4.0.9	19
	CVE-2018-17795	Heap Overflow	[3.9.3, 4.0.9] (20)	[2010.6, 2017.11]	8.8 HIGH	4.0.9	6
	CVE-2018-18557	Heap Overflow	[3.9.3, 4.0.9] (20)	[2010.6, 2017.11]	8.8 HIGH	4.0.9	17
jasper	CVE-2016-9560	Stack Overflow	[1.900.8, 1.900.29] (22)	[2016.10, 2016.11]	7.8 HIGH	< 1.900.30	0
	CVE-2017-14132	Heap Overflow	[1.900.8, 2.0.16] (41)	[2016.10, 2019.3]	6.5 MEDIUM	2.0.13	40
	CVE-2018-19540	Integer Overflow to OOB	[1.900.8, 2.0.16] (41)	[2016.10, 2019.3]	8.8 HIGH	2.0.14	40
	CVE-2018-19541	Heap Overflow	[1.900.8, 2.0.16] (41)	[2016.10, 2019.3]	8.8 HIGH	2.0.14	40
SUM	30		500[2]				330

[1] There are 4 unpatched versions of `lame` for CVE-2017-15046 that are unable to be tested with ASAN, so they are excluded from the dataset.

[2] As illustrated in Table 2, the 500 unpatched versions consist of 30 reference versions and 470 target versions.

Table 7: Comparison results with AFL and AFLGo on PoC Migration.

Software	CVE	Target Version	AFL	AFLGO	VulScope
zziplib	CVE-2018-6381	0.13.63	08m 17s	13m 03s	05m 05s
		0.13.64	00m 58s	15m 47s	04m 59s
		0.13.65	01m 12s	03m 24s	06m 00s
		0.13.66	01m 21s	03m 25s	05m 14s
		0.13.67	01m 33s	02m 54s	07m 50s
audiofile	CVE-2017-6836	0.2.7	✗	✗	✗
	CVE-2017-6831	0.2.7	02h 34m 12s	01h 10m 38s	01m 21s
		0.3.0	✗	✗	00m 52s
		0.3.1	05h 23m 16s	06h 20m 09s	01m 38s
		0.3.2	06h 01m 24s	35m 11s	01m 37s
		0.3.3	02h 03m 00s	01h 18m 51s	01m 37s
		0.3.4	06h 33m 58s	04h 27m 04s	01m 38s
lame	CVE-2015-9101	3.97	✗	✗	✗
		3.98	04m 12s	07m 13s	01m 40s
		3.98.2	03m 57s	06m 25s	01m 00s
		3.98.4	05m 04s	18m 58s	01m 06s
	CVE-2017-15018	3.97	✗	✗	✗
	CVE-2017-15045	3.97	✗	✗	✗
		3.98	01m 01s	00m 19s	00m 35s
		3.98.2	01m 11s	00m 19s	00m 29s
		3.98.4	01m 23s	00m 45s	00m 27s
jasper	CVE-2016-9560	1.900.26	✗	✗	02m 08s
		1.900.27	✗	✗	02m 15s
		1.900.28	✗	✗	02m 22s
		1.900.29	✗	✗	02m 17s
libtiff	CVE-2016-10092	3.9.3	00m 19s	00m 13s	00m 20s
		3.9.4	00m 19s	00m 11s	00m 24s
		3.9.5	00m 06s	00m 06s	00m 18s
		3.9.6	00m 06s	00m 05s	00m 18s
		3.9.7	00m 06s	00m 06s	00m 18s
		4.0.0alpha5	00m 13s	00m 17s	01m 18s
		4.0.0alpha6	00m 12s	00m 17s	00m 41s
	CVE-2018-12900	4.0.0alpha4	01m 10s	00m 14s	00m 13s
		4.0.0alpha5	01m 14s	00m 19s	00m 17s
		4.0.0alpha6	01m 19s	00m 18s	00m 16s
	CVE-2016-10269	4.0.0alpha4	02h 59m 23s	38m 41s	15m 25s
		4.0.0alpha5	02h 15m 34s	53m 56s	12m 30s
		4.0.0alpha6	02h 07m 12s	51m 55s	36m 35s
		4.0.0beta7	05m 39s	01m 32s	59m 15s
		4.0.0	05m 52s	01m 11s	36m 02s
		4.0.1	05m 42s	00m 56s	31m 49s
		4.0.2	05m 51s	00m 56s	38m 25s
		4.0.3	05m 32s	00m 59s	30m 33s
		4.0.4	05m 32s	00m 59s	31m 08s
		4.0.4beta	05m 14s	01m 02s	31m 28s
		4.0.5	05m 09s	01m 00s	31m 59s
		4.0.6	04m 38s	01m 02s	27m 26s

Software	CVE	Target Version	AFL	AFLGO	VulScope
libtiff	CVE-2016-10095	3.9.3	✗	✗	✗
		3.9.4	✗	✗	✗
		3.9.5	✗	✗	✗
		3.9.6	✗	✗	✗
		3.9.7	✗	✗	✗
	CVE-2016-10093	3.9.3	10m 04s	08m 54s	02m 56s
		3.9.4	09m 46s	09m 32s	03m 13s
		3.9.5	01m 38s	01m 38s	04m 23s
		3.9.6	01m 40s	02m 00s	04m 55s
		3.9.7	02m 26s	01m 41s	04m 45s
		4.0.0alpha4	32m 43s	13m 26s	03m 44s
		4.0.0alpha5	34m 16s	13m 11s	04m 05s
		4.0.0alpha6	02h 18m 35s	01h 16m 01s	33m 14s
	CVE-2018-17795	3.9.3	✗	✗	✗
		3.9.4	✗	✗	✗
		3.9.5	✗	✗	✗
		3.9.6	✗	✗	✗
		3.9.7	✗	✗	✗
		4.0.0alpha4	✗	✗	✗
		4.0.0alpha5	✗	✗	✗
		4.0.0alpha6	✗	✗	✗
		4.0.0	✗	✗	✗
		4.0.1	✗	✗	✗
		4.0.2	✗	✗	✗
		4.0.3	✗	✗	✗
		4.0.4	✗	✗	01m 03s
		4.0.4beta	✗	✗	01m 12s
		4.0.5	✗	✗	01m 01s
		4.0.6	✗	✗	01m 03s
		4.0.7	✗	✗	01m 03s
		4.0.8	✗	✗	00m 44s
	CVE-2018-18557	3.9.3	✗	✗	✗
		3.9.4	✗	✗	✗
		4.0.0alpha4	✗	✗	13m 23s
		4.0.0alpha5	✗	✗	12m 53s
		4.0.0alpha6	✗	✗	13m 43s
		4.0.0beta7	✗	✗	25m 27s
		4.0.0	✗	✗	14m 25s
		4.0.1	✗	✗	30m 37s
		4.0.2	✗	✗	22m 13s
		4.0.3	✗	✗	05m 33s
		4.0.4	✗	✗	14m 10s
		4.0.4beta	✗	✗	05m 57s
		4.0.5	✗	✗	14m 39s
		4.0.6	✗	✗	05m 28s
		4.0.7	✗	✗	14m 10s
		4.0.8	✗	✗	14m 59s

1) This experiment is conducted on all 94 affected versions that need PoC migration (illustrated in Table 2).
2) ✗: fail to achieve PoC Migration in 8 hours

Table 8: Comparison results with AFL and AFLGo on the number of seeds that reach the buggy site.

Software	CVE	Target Version	AFL	AFLGO	VulScope
zziplib	CVE-2018-6381	0.13.63	10	11	2
		0.13.64	1	2	2
		0.13.65	2	2	1
		0.13.66	2	2	1
		0.13.67	2	2	1
audiofile	CVE-2017-6831	0.2.7	3	1	1
		0.3.0	0(✗)	0(✗)	1
		0.3.1	2	1	1
		0.3.2	1	1	1
		0.3.3	2	5	1
		0.3.4	2	4	1
lame	CVE-2015-9101	3.98	2	2	1
		3.98.2	1	2	1
		3.98.4	1	2	2
	CVE-2017-15045	3.98	1	2	1
		3.98.2	1	1	1
		3.98.4	1	1	3
libtiff	CVE-2016-10092	3.9.3	3	5	2
		3.9.4	5	5	3
		3.9.5	2	3	5
		3.9.6	2	2	5
		3.9.7	2	3	5
		4.0.0alpha5	8	8	1
		4.0.0alpha6	8	8	1
	CVE-2016-10269	4.0.0alpha4	1	1	1
		4.0.0alpha5	1	1	2
		4.0.0alpha6	1	1	3
		4.0.0beta7	1	1	1
		4.0.0	1	1	1
		4.0.1	1	1	1
		4.0.2	1	1	1
		4.0.3	1	1	1
		4.0.4	1	1	1
		4.0.4beta	1	1	1
		4.0.5	1	1	1
		4.0.6	1	1	1
libtiff	CVE-2016-10093	3.9.3	2	2	2
		3.9.4	2	2	2
		3.9.5	1	1	5
		3.9.6	1	1	5
		3.9.7	1	1	5
		4.0.0alpha4	1	1	1
		4.0.0alpha5	1	1	1
		4.0.0alpha6	1	1	2
	CVE-2018-17795	4.0.4	250(✗)	161(✗)	3
		4.0.4beta	129(✗)	42(✗)	3
		4.0.5	118(✗)	157(✗)	3
		4.0.6	270(✗)	162(✗)	3
		4.0.7	66(✗)	60(✗)	3
		4.0.8	81(✗)	168(✗)	1
	CVE-2018-18557	4.0.0alpha4	0(✗)	0(✗)	1
		4.0.0alpha5	0(✗)	0(✗)	2
		4.0.0alpha6	0(✗)	0(✗)	1
		4.0.0beta7	0(✗)	0(✗)	2
		4.0.0	0(✗)	0(✗)	2
		4.0.1	0(✗)	0(✗)	2
		4.0.2	0(✗)	0(✗)	2
		4.0.3	0(✗)	0(✗)	1
		4.0.4	0(✗)	0(✗)	2
		4.0.4beta	0(✗)	0(✗)	1
		4.0.5	0(✗)	0(✗)	2
		4.0.6	0(✗)	0(✗)	1
		4.0.7	0(✗)	0(✗)	2
		4.0.8	0(✗)	0(✗)	3
	CVE-2018-12900	4.0.0alpha4	8	3	6
		4.0.0alpha5	5	6	4
		4.0.0alpha6	8	6	4
jasper	CVE-2016-9560	1.900.26	0(✗)	0(✗)	1
		1.900.27	0(✗)	0(✗)	1
		1.900.28	0(✗)	0(✗)	1
		1.900.29	0(✗)	0(✗)	1

1) This experiment is conducted on all target versions that are successfully migrated by VulScope to help understand its superiority.
2) We record the number of the seeds that can reach the buggy site until the success of migration. For those versions that failed to be migrated by AFL/AFLGO, we record the number of the seeds that can reach the buggy site within the time limit of 8 hours.
3) ✗: fail to achieve PoC migration in 8 hours