TransferFuzz: Fuzzing with Historical Trace for Verifying Propagated Vulnerability Code

Siyuan Li[1,2], Yuekang Li[3], Zuxin Chen[1,2], Chaopeng Dong[1,2], Yongpan Wang[1,2]
Hong Li[1,2,*], Yongle Chen[4], Hongsong Zhu[1,2]

[1]*Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China*
[2]*School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China*
[3]*School of Computer Science and Engineering, The University of New South Wales, Sydney, Australia*
[4]*College of Computer Science and Technology, Taiyuan University of Technology, Shanxi, China*

lisiyuan@iie.ac.cn, yuekang.li@unsw.edu.au, {chenzuxin,dongchaopeng,wangyongpan,lihong}@iie.ac.cn,
chenyongle@tyut.edu.cn, zhuhongsong@iie.ac.cn

Abstract—Code reuse in software development frequently facilitates the spread of vulnerabilities, making the scope of affected software in CVE reports imprecise. Traditional methods primarily focus on identifying reused vulnerability code within target software, yet they cannot verify if these vulnerabilities can be triggered in new software contexts. This limitation often results in false positives. In this paper, we introduce TransferFuzz, a novel vulnerability verification framework, to verify whether vulnerabilities propagated through code reuse can be triggered in new software.

Innovatively, we collected runtime information during the execution or fuzzing of the basic binary (the vulnerable binary detailed in CVE reports). This process allowed us to extract historical traces, which proved instrumental in guiding the fuzzing process for the target binary (the new binary that reused the vulnerable function). TransferFuzz introduces a unique Key Bytes Guided Mutation strategy and a Nested Simulated Annealing algorithm, which transfers these historical traces to implement trace-guided fuzzing on the target binary, facilitating the accurate and efficient verification of the propagated vulnerability.

Our evaluation, conducted on widely recognized datasets, shows that TransferFuzz can quickly validate vulnerabilities previously unverifiable with existing techniques. Its verification speed is 2.5 to 26.2 times faster than existing methods. Moreover, TransferFuzz has proven its effectiveness by expanding the impacted software scope for 15 vulnerabilities listed in CVE reports, increasing the number of affected binaries from 15 to 53. The datasets and source code used in this article are available at https://github.com/Siyuan-Li201/TransferFuzz.

Index Terms—Software Security, Binary Analysis, Vulnerability Verification

I. INTRODUCTION

Code reuse is an important part of modern software development, providing benefits in terms of efficiency and functionality. A vast array of open-source code and third-party libraries are available on platforms like GitHub [1], SourceForge [2] and Vcpkg [3]. These resources enable developers to incorporate existing code into new projects, allowing them to concentrate on developing unique software features. Synopsys' report [4] in 2023 shows that 97% of audit software reuses the code of at least one third-party library, highlighting the prevalence of code reuse in development practices.

Despite its benefits, code reuse can introduce significant risks. The widespread use of shared code can propagate vulnerabilities across numerous software projects [5]. While reused code benefits from iterative updates, it is not immune to flaws, which can spread as the code is reused. Security researchers typically report new vulnerabilities to platforms like CVE [6] or NVD [7]. However, they may not fully consider the scope of the software affected by these vulnerabilities through code reuse. This paper mainly focuses on the vulnerability propagation verification of C/C++ binary, an area where the impact of vulnerabilities can extend beyond what is initially reported [5].

Reflecting their distinct motivations, we categorize existing methods into three main types: Code Reuse Detection methods, Patch Presence Detection methods, and Directed Fuzzing methods. These methods can all detect the propagation of vulnerabilities due to code reuse.

Firstly, Code Reuse Detection methods aim to detect the presence of vulnerable functions in the target software. OS-SPolice [8], B2SFinder [9] and FirmSec [10] use constant features such as strings, arrays, and jump tables to detect code reuse and correlate the vulnerabilities. Besides, Gemini [11], JTrans [12], and Asteria [13] perform function similarity matching to detect vulnerabilities. Moreover, LibAM [5], LibDI [14], and TPLite [15] use code areas or function ratios from third-party libraries to detect propagated vulnerabilities. All these approaches focus on detecting the presence of the vulnerable code in the target software and don't further confirm whether the vulnerability poses a threat (**P1**). For instance, these vulnerable functions may have been patched.

Secondly, Patch Presence Detection methods aim to detect whether the reused vulnerability functions in the target software are patched. Fiber [16] and PDiff [17] use symbolic execution methods to detect whether the code in the target software is closer to the vulnerable code or the patched code. LibvDiff [18] uses function call paths to detect fine-grained

versions of the target software, thus confirming whether the reused code is a patched version. However, recent research shows that both of these methods only detect whether the vulnerability is patched rather than judge whether the vulnerability can be triggered in the new software context [19]. This discrepancy has led to a large number of false positives [5] (**P1**). There are two main reasons: the vulnerable function may be unreachable and the critical variables may be uncontrollable, which is described in detail in Section II.

Thirdly, some researchers wanted to use directed fuzzing [20]–[23] for Patch Testing and Crash Production in software. However, the performance of current directed fuzzing approaches has been less than optimal. They often require several hours to verify a single vulnerability and struggle to trigger vulnerabilities necessitating complex logic [21] (**P2**). To our knowledge, there is only one work OCTOPOCs [24] that uses historical knowledge from the Proofs-of-Concept (POCs) of third-party libraries like us and tries to trigger vulnerabilities in the new software. However, OCTOPOCs exhibit certain limitations. It is only suitable for small binaries to avoid path explosion and requires a specific input file for the target binary consistent with the basic binary (**P3**). These limitations are discussed in detail in Section II.

In summary, the existing methods suffer from the following problems:

- **P1**: Code Reuse Detection and Patch Presence Detection methods fail to confirm that the vulnerability is triggerable in the target software, thus leading to false positives.
- **P2**: Directly using directed fuzzing techniques to verify vulnerabilities is slow and difficult to trigger vulnerabilities with complex logic.
- **P3**: OCTOPOCs is limited by specific types of target binaries or input files, and cannot verify vulnerabilities in large binaries.

To overcome these challenges, we propose TransferFuzz, a novel vulnerability verification framework. We found that by collecting some runtime information of the **basic binary** (the vulnerable binary detailed in CVE reports), we can extract the historical traces to guide the fuzzing process for the **target binary** (new binary that reused the vulnerable function of basic binary). By using historical traces, we can achieve faster triggering of vulnerabilities in new software and trigger complex logic vulnerabilities (for **P2**). By generating POCs for new software, it is possible to automatically verify the vulnerability is an actual threat to the new software (for **P1** and **P3**).

TransferFuzz is not a new directed fuzzing technique, but a general framework designed to enhance existing directed fuzzing. Our framework utilizes runtime features to generate two types of historical traces: function-level traces and key-bytes traces. Different from Directed Fuzzing, we proposed a **Trace-guided Fuzzing** framework that incorporates a Key Bytes Guided Mutation strategy and a Nested Simulated Annealing algorithm to fuzz the target binary. This innovative approach allows for more accurate and efficient vulnerability verification. The motivation is detailed in Section II.
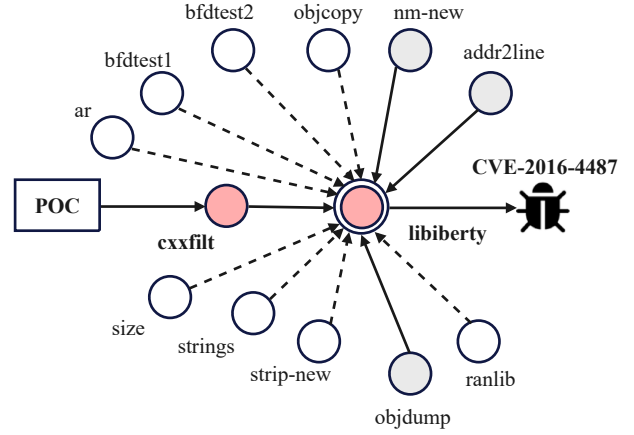

Fig. 1. Propagated vulnerability code. Red nodes are vulnerability information in CVE reports and historical research, white nodes are binaries that reuse vulnerable code, and gray nodes are binaries affected by the vulnerability.

Our evaluation, leveraging widely utilized datasets introduced by AFLGo and OCTOPOCs, demonstrates the capability of TransferFuzz to validate vulnerabilities that were previously unverifiable through existing techniques, achieving a verification speed that is 2.5 to 26.2 times faster than existing methods. Furthermore, TransferFuzz expands the impacted software scope of 15 vulnerabilities in the CVE report from 15 binaries to 53, underscoring its effectiveness in verifying propagated vulnerability code.

The contributions of this paper are as follows:

- We propose TransferFuzz, the first vulnerability verification framework based on fuzzing to accurately and efficiently verify propagated vulnerability code.
- TransferFuzz provides strong evidence for vulnerability propagation by utilizing the historical traces and triggering the vulnerability.
- We analyzed the software scope impacted by the 15 vulnerabilities in the CVE report and identified 38 additional affected binaries not previously reported.

II. MOTIVATION

We illustrate our approach with several examples. Figure 1 depicts the vulnerability CVE-2016-4487 in *libiberty* as described in CVE [6] and NVD [7] reports. AFLGo [20] employs fuzzing on *cxxfilt* and generates a Proof of Concept (POC) that triggered this vulnerability. Subsequent analyses [23], [25] assumed that CVE-2016-4487 exclusively affected the *cxxfilt* binary, ignoring potential impacts on other binaries. After manual analysis, we discovered that 11 new binaries reuse the vulnerable *libiberty* code. However, since the input file formats required by these binaries are different, their POC cannot be universal, and it is difficult to determine which binaries can trigger this vulnerability. By manually constructing appropriate inputs, we successfully triggered the vulnerability in *objdump*, *nm-new*, and *addr2line*. This example demonstrates that vulnerabilities in C/C++ binaries can propagate through code reuse, and the scope of software
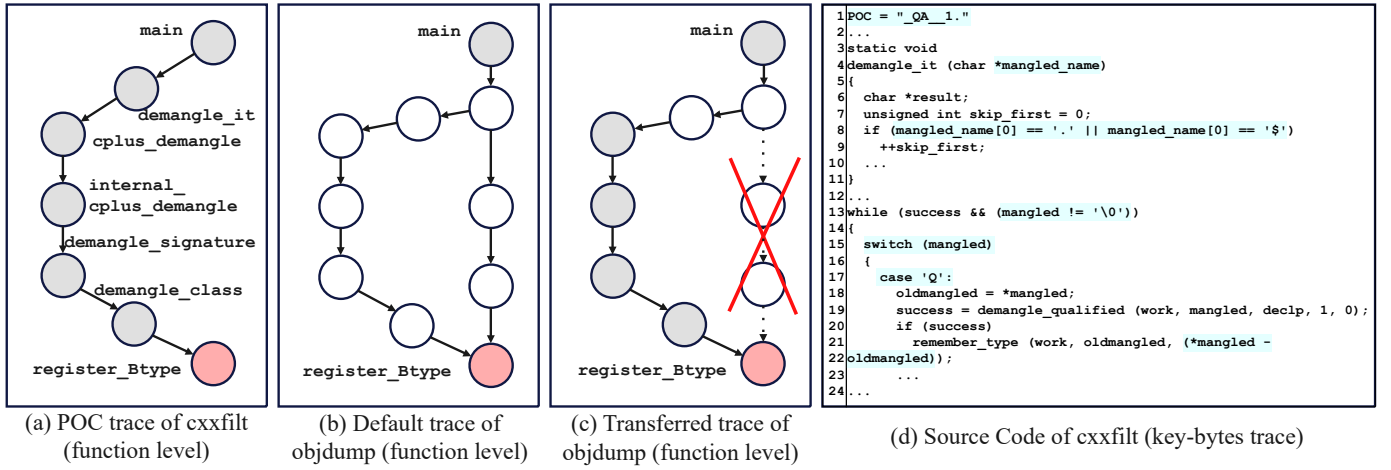
```
1  POC = "_QA__1."
2  ...
3  static void
4  demangle_it (char *mangled_name)
5  {
6    char *result;
7    unsigned int skip_first = 0;
8    if (mangled_name[0] == '.' || mangled_name[0] == '$')
9      ++skip_first;
10   ...
11 }
12 ...
13 while (success && (mangled != '\0'))
14 {
15   switch (mangled)
16   {
17     case 'Q':
18       oldmangled = *mangled;
19       success = demangle_qualified (work, mangled, declp, 1, 0);
20       if (success)
21         remember_type (work, oldmangled, (*mangled -
22 oldmangled));
23       ...
24 ...
```

| (a) POC trace of cxxfilt (function level) | (b) Default trace of objdump (function level) | (c) Transferred trace of objdump (function level) | (d) Source Code of cxxfilt (key-bytes trace) |

Fig. 2. A motivation example. The red node is the vulnerability function of CVE-2016-4487. Gray nodes are functions known to be on the vulnerability-triggering path, and white nodes are functions that may pass through. The blue code in 1(d) is the code in *cxxfilt* that accesses POC related bytes

affected by a vulnerability extends beyond what vulnerability reports or prior research suggest.

However, not every instance of reused vulnerability code results in vulnerability propagation. As depicted in Figure 1, eight binaries remained unaffected by CVE-2016-4487. Despite containing the vulnerable code, the vulnerable functions were not reachable within those binaries. Expect the reachable, another factor also plays an important role. For instance, CVE-2016-10095 describes a stack-based buffer overflow vulnerability in *LibTIFF*. When the value of the second parameter 'tag' of the function *_TIFFVGetField* is 0x13d, it will cause a buffer overflow. Although *OpenJPEG* incorporates the vulnerable *LibTIFF* code, the vulnerability does not threaten *OpenJPEG* because the critical variable 'tag' is hard-coded, rendering the vulnerability's critical variables uncontrollable. Both Code Reuse Detection methods and Path Presentation Detection methods fail to confirm if the vulnerability functions are reachable or if critical variables are controllable (**P1**).

Besides, in our evaluation of *objdump*, we observed that handling large binaries could lead to path explosion, causing methods like OCTOPOCs [24] to fail. Another challenge arises with vulnerabilities like CVE-2017-8393, an out-of-bounds read issue. OCTOPOCs struggles with scenarios where the crash cause does not align with specific bytes in a POC. In fact, OCTOPOCs can only handle special scenarios where the inputs to the basic software and the target software contain the full crash-cause byte and are processed in the same way (**P3**). It is worth noting that most of software in its dataset takes PDF files or ZIP files as input [24].

Our Approach FuzzGuard [26] explores the use of historical fuzzing data within the same software to guide new fuzzing, and VulScope [27] explores the use of historical fuzzing data from different versions of the software to guide new fuzzing. Inspired by these two efforts, we propose leveraging historical fuzzing data from different software that share reused code areas to guide fuzzing (for **P2**). Our novel approach involves executing POCs from the basic binary to obtain historical traces that guide the fuzzing process in a

new target binary to trigger the vulnerability (for **P1** and **P3**). Specifically, we focus on two types of historical traces: function-level traces and key-bytes traces.

Function-level Traces: As demonstrated in Figure 2(a), executing the *cxxfilt* for CVE-2016-4487 with POC reveals the function call sequence that triggers the vulnerability, which is treated as function-level traces. This function-level trace allows us to move beyond the blind fuzzing for *objdump* depicted in Figure 2(b), adopting a more targeted approach as shown in Figure 2(c). This method only focuses on paths relevant to the vulnerability, thus speeding up the fuzzing process. Through static analysis, we found that there are 3150 different function calling paths in *objdump* from the main function to the vulnerability function. However, we identified 42 paths by function-level traces of *cxxfilt* and discarded the rest.

Key-bytes Traces: Figure 2(d) shows how certain key bytes in the *cxxfilt* POC can help bypass branch condition constraints in reused code (The highlighted code in the figure). These branches are also included in the target binary's reused code, hindering fuzzing speed. Extracting these bytes as key-bytes traces can significantly accelerate the fuzzing process, enhancing the efficiency of vulnerability verification.

III. METHODOLOGY

A. Overview

The workflow of TransferFuzz, presented in Figure 3, consists of three modules: Function-level Traces Extraction, Key-bytes Traces Extraction, and Trace Guided Fuzzing. We assume that Code Reuse Detection methods such as LibAM [5] have been used to detect that the target binary reuses the vulnerable code in the basic binary, and we want to further verify that the vulnerability can be triggered in the target binary. We take a basic binary with its POC and the target binary as inputs. Then, TransferFuzz judges whether the target binary is impacted by the vulnerability of the basic binary and generates a POC for the target binary if it is impacted.

In the Function-level Traces Extraction module, we extract Function Call Sequence (FCS) from basic binary. Firstly, we
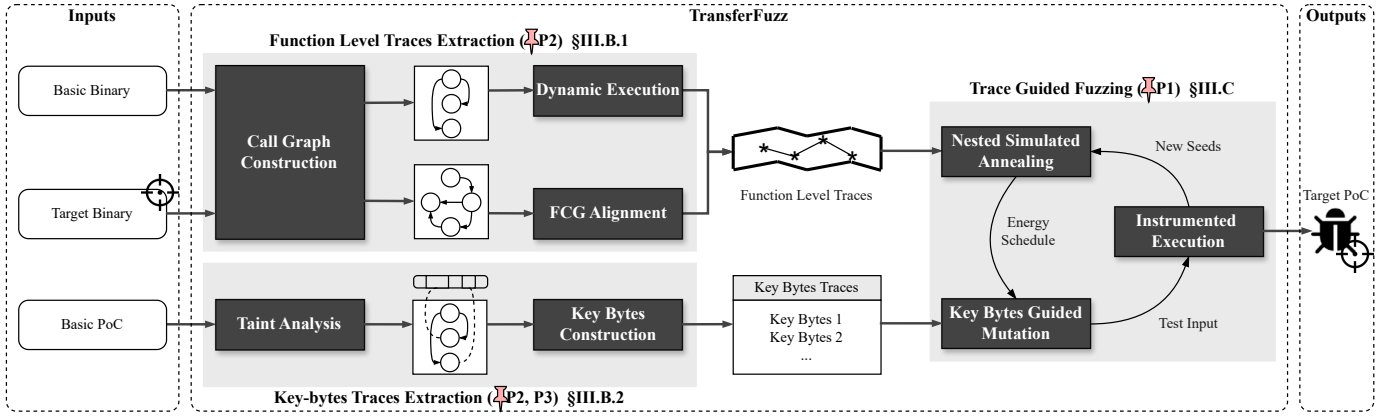
Fig. 3. The workflow of TransferFuzz.

execute the corresponding POC on the basic binary to collect run-time FCS as function-level traces to guide the subsequent fuzzing of the target binary. In addition, we find the path that triggers a vulnerability may not be single. Therefore, we also run a generic directed fuzz on the basic binary to collect more paths that may trigger the vulnerability. All the paths with run-time information that triggered the vulnerability are function-level traces and are taken as input for the final step.

In the Key-bytes Traces Extraction module, we use taint analysis technology to extract the bytes in the POC that directly affect the conditional judgment statements in the reused code. These bytes are then conducted as a dictionary to guide the mutation of the target binary. In this way, some complex branch constraints can be bypassed when fuzzing the target binary.

In the Trace Guided Fuzzing module, we transfer historical traces to the target software to efficiently fuzz and thus verify vulnerabilities. During the Fuzzing process, in addition to the general Fuzzing mutation strategy, we propose the Key Bytes Guided Mutation strategy to use key-bytes traces to mutate seeds. Besides, we propose a Nested Simulated Annealing algorithm. In detail, we record the key jump points of vulnerability paths in function-level traces to form a state machine. For each test case, we schedule energy based on its execution state and the current most desired state to explore. Our goal is to continuously explore the most desired state path and avoid missing other possible paths as much as possible.

Finally, there are three possible results: the vulnerability is triggered, the reuse code is reached but the vulnerability is not triggered, and the reuse code area is not reached. For the first result, we consider the propagated vulnerability confirmed. For the latter two results, we consider that the vulnerability cannot be or is difficult to be triggered in the target software, and further manual verification is required. Compared with existing methods, although TransferFuzz has trouble making judgments about target binary without a crash, TransferFuzz can provide strong evidence for the propagation of the vulnerability that the POC triggers the vulnerability. The experimental results also further prove the vulnerability verification capability of TransferFuzz.

B. Historical Trace Extraction

In this section, our goal is to extract historical traces from basic binary. We found two types of runtime information that help a lot in fuzzing the target binary. One is the function call sequence (FCS) for the basic binary, the other is key bytes in POC.

1) Function-level Traces Extraction: Firstly, we aim to extract function call sequence (FCS) from basic binary. Target binary and basic binary can often trigger vulnerabilities along the same path due to having the same reused code. Therefore, we would like to collect the vulnerability-triggering path of the basic binary and have the target binary fuzzing along this path. We have used two ways to extract FCS together. The first is to directly use the basic binary to execute POC to record run-time information, and the second is to use generic directed fuzzing on the basic binary to generate new POCs and record run-time information.

Directly Execute POC: For the basic binary whose vulnerability information with POCs, it can be directly executed POC to collect run-time information. Directly executing is very helpful when the vulnerability has the execution of complex logic and is not well triggered through generic fuzzing. We want to collect the function call sequence that triggered the vulnerability, and we want the target binary to run along these sequences during the fuzzing process, thus excluding irrelevant paths and enhancing the process's focus and efficiency. We run the basic binary and log the function call stack when the basic binary crashes.

Fuzzing Basic Binary: POCs can be generated using generic directed fuzzing for basic binary without POCs in the vulnerability information. Besides, the basic binary with POCs can be expanded with more possible paths. In fact, using only one vulnerability triggering path may make TransferFuzz's function-level traces limited. In extreme cases, if the paths in the function-level traces are not the most easily triggered ones, it will affect the fuzzing of the target binary instead. Therefore, it is necessary to collect sufficiently rich paths for the basic binary by employing fuzzing in advance. This is also an offline process and does not affect the vulnerability verification efficiency of TransferFuzz because it only needs to be executed once on the basic binary to detect all possible

propagation of the target binary. We use the state-of-the-art directed fuzzing method SelectFuzz to perform directed fuzzing on the basic binary for crash reproduction, which is one of the tasks that directed fuzzing specializes in. By setting the source code line where the vulnerability is located, Fuzzer can calculate the distance from the current test case to the target code line, and guide the seed to mutate in the direction close to the vulnerability.

We do this based on such a motivation that the basic binary and target binary tend to have the same vulnerability triggering path in the reused code area due to having the same code. The vulnerability paths that are easy to reach during Fuzzing in the basic binary are also easier to reach in the target binary.

FCG Alignment: Finally, we need to align the basic binary's function-level traces with the target binary. We found that many target binaries make minor changes to the reused code. Therefore, we need to align the functions in the basic binary with those in the target binary using function matching. By doing this, the function-level traces can be directly applied to the target binary.

The easiest way to do this is to use the function names directly. This works when the target binary is not stripped. However, in practical scenarios, where some binary may be considered strip to remove function names, we can use Binary Code Similarity Detection (BCSD) methods for function matching. We used LibAM [5] in our evaluation to perform function matching when function names were missing after stripping. This step can be easily replaced by the existing BCSD methods [12], [13].

After the function matching, we can obtain a sub-path of the path in the history function call sequence of the basic binary. The original path reaches the vulnerability function from the main function of the basic binary, while the sub-path may be a path in the target binary that starts from one of the nodes in the original path and reaches the vulnerability function. We record the sub-path, which can directly guide the target binary's fuzzing.

2) Key-bytes Traces Extraction: In addition to function call sequences, we found that some bytes of the POC may affect key condition variables in the code. If these key bytes in the basic binary are extracted, it can help the fuzzing process of the target binary. This idea is similar to that of OCTOPOCs, but it is worth noting that OCTOPOCs just extract the key bytes in the POC and splice them together to generate a new POC, which is easy to fail. Instead, we hope to use these key bytes as a dictionary to assist the target binary's mutation process.

We implement this process using taint analysis techniques. Specifically, we mark the POC of the basic binary as tainted, and during execution, we pay attention to whether the comparison variables of the conditional statements in the reused code area come from the POC. Then, we compare the bytes accessed by reused code with bytes in POC to extract the bytes that are directly passed unchanged from the POC as key-bytes traces. The specific process is detailed in Algorithm 1. During the target binary's fuzzing process, when these tainted

---

Algorithm 1: Key Bytes Extraction Algorithm

Input: binaryExecutable B, reusedFunctionList F, proofOfConceptFile P

Output: accessedBytes A

1 **Function** ExtractKeyBytes(B, F, P):
2     $PBytes \leftarrow$ ReadBytes(P);
3     $BImages \leftarrow$ LoadImages(B);
4   **foreach** $image \in BImages$ **do**
5   **foreach** $func \in F$ **do**
6   **if** $func$ is in $image$ **then**
7   InstrumentEntryAndExit($func$);

8     $A \leftarrow \emptyset$;
9   **for** *each instruction I in B* **do**
10   **if** *I is MemoryAccess* **and** *IsInFunctionList(I, F)* **then**
11             $address \leftarrow$ GetAccessAddress(I);
12   **if** *$address$ corresponds to $PBytes$* **then**
13                 $byte \leftarrow$ GetByteAt($address$, $PBytes$);
14                 $A \leftarrow A \cup \{(address, byte)\}$;

15   **return** A;

---

bytes are inserted at specific locations, it is easy to satisfy the corresponding conditional judgment in the reused code.

C. Trace Guided Fuzzing

In this section, our goal is to transfer the extracted historical trace to the target binary and efficiently verify the propagated vulnerability code in the target binary using Fuzzer with the transferred traces.

1) Nested Simulated Annealing: Unlike directed fuzz, which sets only one target line of code, TransferFuzz wants to utilize function call sequences from historical trace to schedule more energy to seeds exploring along vulnerability paths and less energy to seeds on unrelated paths.

Initially, a unique state machine is established for every execution path, meticulously recording every function encountered along each path throughout the fuzzing process. To manage the complexity of historical traces, distinct state machines are assigned to individual paths. Within these state machines, each state signifies a function node, charting a sequential path from the entry function of the reused code to the function harboring the vulnerability. Energies are judiciously scheduled to test cases based on a comparison between their achieved states and the latest state documented by the corresponding state machine, prioritizing those deemed most relevant.

Our objective is to preferentially allocate energies to the most recently identified states, while avoiding the pitfall of exclusively advancing along the deepest path, to the neglect of other potentially vulnerable paths. To achieve this balanced approach, we adopt the simulated annealing algorithm, as delineated in AFLGo. Simulated Annealing (SA) [28] is inspired by the annealing process in metallurgy, a technique

Algorithm 2: Nested Simulated Annealing Algorithm

Input: current state cur, states list $states$, start time list $time$, current time t

Output: energy E

1 **Function** CalcEnergy(cur, $states$, $time$, t):
2   **if** cur is new **then**
3   NewState(t, $states$, $time$);
4   **return** 0.5;
5    $idx \leftarrow$ Index of cur in $states$;
6    $start \leftarrow time[idx]$;
7    $temp \leftarrow$ SimulatedAnnealing($start$, t);
8    $E \leftarrow 1 - 0.5 \times temp$;
9   **for** $i = idx + 1$ **to** length of $states$ **do**
10       $newStart \leftarrow time[i]$;
11       $newTemp \leftarrow$ SimulatedAnnealing($newStart$, t);
12       $E \leftarrow E \times 0.5 \times newTemp$;
13   **return** E;
14 ;
15 **Function** SimulatedAnnealing($start$, t):
16    $T \leftarrow$ Cooling(t, $start$);
17   **return** T;

Fig. 4. State machine diagram in NSA algorithm.

$$
E_j = \begin{cases} \prod_{i>j} 0.5T_i & \text{if } j = 0, \\ (1 - 0.5T_j) \cdot \prod_{i>j} 0.5T_i & \text{if } j > 0. \end{cases} \quad (1)
$$

Within this context, E_j denotes the energy of the j th state, while T_j represents the temperature parameter [20] for the j th state, calculated using the simulated annealing algorithm.

We present the detailed procedure of the nested simulated annealing algorithm using the example in Figure 4. Initially, TransferFuzz operates like a generic directed fuzzing tool before reaching the reused code area. At this stage, energy scheduling is a basic simulated annealing strategy [20]. However, once *state0* of the state machine is encountered, we trigger the first simulated annealing algorithm. This algorithm allocates half of the energy (0.5) to seeds that have achieved *state0* and distributes the remaining half to those that have not. Over time, the energy for *state0* escalates to 1, diminishing the energy for seeds failing to reach *state0* to zero. Upon the initial reach of the new *state1*, the second simulated annealing algorithm kicks in, assigning half of the energy (0.5) to seeds at *state1* while the other 0.5, including those stuck at *state0*, share the rest. This 0.5 split still adheres to the principles of the first algorithm. A new simulated annealing algorithm is activated whenever a new state is reached. By using this nested simulated annealing algorithm, the energy of the seeds that are not related to the vulnerability paths in the historical trace decay rapidly and mutates in the direction that drives the state machine forward.

2) Key Bytes Guided Mutation: TransferFuzz adopts the mutation strategy of traditional directed fuzzing but enhances it significantly. TransferFuzz distinguishes itself by utilizing key bytes extracted from the binary's historical traces to guide test case generation. Specifically, each unique sequence of consecutively accessed bytes in memory is treated as a row in the dictionary. By systematically integrating these key bytes into a dictionary and applying it at every mutation phase, TransferFuzz can bypass some difficult-to-implement branch constraints and demonstrably boost fuzzing efficiency, as supported by our experimental results.

Note that the KBGM algorithm, while similar to the taint analysis used in AFL++ [29], Windranger [23], and REDQUEEN [30], is more robust. It extracts knowledge from another binary and can handle branches with complex conditions. This capability allows it to handle complex vulnerabilities, as demonstrated in the evaluation.

IV. IMPLEMENTATION

We implement the TransferFuzz prototype with over 2000 lines of Python and C code. TransferFuzz is a generalized transferred trace-based fuzzing framework that can based on

that involves heating and controlled cooling of a material to increase its crystal size and reduce its defects. In AFLGo, a designated moment, denoted as tx, signifies the transition from a phase of extensive exploration to a focused exploitation phase. Intuitively, at the juncture of tx, the behavior of the simulated annealing algorithm mirrors that of a classical gradient descent algorithm, effectively embodying a greedy search approach. Before reaching tx, the algorithm ensures a thorough exploration of alternative paths, thus maintaining a holistic assessment of potential vulnerabilities.

Unlike AFLGo's singular Markov chain approach, we model each execution path as an independent state machine, with each state node representing a function node on the path. For each state machine, we maintain a time list, documenting the initial encounter with each state. This approach converts the task of energy scheduling for each state into determining their joint probability distribution. Upon the emergence of a new state, we initially set the energy distribution between the new and existing states to 0.5. This ratio is dynamically adjusted via the simulated annealing algorithm, incrementally favoring the newer states with increased energy while reducing that of the older states. Such a strategy facilitates the individualized management of simulated annealing timelines for each state, thereby ensuring the state machine collectively gravitates towards the most recent state. The pseudo-code for the algorithm is given in Algorithm 2.

In detail, for each input seed, the energy is calculated based on the state reached during the execution. The calculation of energy for every state is defined as Equation 1:

TABLE I
DATASET SCALE

Vul	Source	Type	CrashFunc	Code Reuse	Propagated	Propagated Binaries
CVE-2016-10095	tiffsplit	CWE-119	_TIFFVGetField	24	2	thumbnail, tiffcmp
CVE-2016-5318	thumbnail	CWE-119	_TIFFVGetField	24	2	tiffsplit, tiffcmp
CVE-2016-4487	cxxfilt	CWE-416	register_Btype	11	3	objdump, nm-new, addr2line
CVE-2016-4489	cxxfilt	CWE-190	string_appendn	11	3	objdump, nm-new, addr2line
CVE-2016-4490	cxxfilt	CWE-190	d_unqualified_name	11	3	objdump, nm-new, addr2line
CVE-2016-4491	cxxfilt	CWE-119	d_print_comp_inner	11	3	objdump, nm-new, addr2line
CVE-2016-4492	cxxfilt	CWE-119	do_type	11	3	objdump, nm-new, addr2line
CVE-2016-6131	cxxfilt	CWE-20	demangle_class_name	11	3	objdump, nm-new, addr2line
CVE-2017-7303	strip	CWE-125	section_match	11	1	objcopy
CVE-2017-11733	swftocxx	CWE-125	stackswap	6	4	swftophp, swftoperl, swftopython, swftotcl
CVE-2018-8807	swftophp	CWE-416	getString	6	4	swftocxx, swftoperl, swftopython, swftotcl
CVE-2018-8962	swftophp	CWE-416	getName	6	4	swftocxx, swftoperl, swftopython, swftotcl
CVE-2017-18267	Poppler	CWE-476	FoFiType1C::getOp	1	1	xpdf
CVE-2018-11102	libav	CWE-119	mov_probe	1	1	ffmpeg
CVE-2018-20330	libjpeg-turbo	CWE-190	__memcpy_avx_unaligned	1	1	mozjpeg

any of the directed fuzzing techniques. We have selected the latest SelectFuzz [31], to implement TransferFuzz.

Historical Trace Extraction To extract function call sequences from basic binary, we use GDB [32] for dynamic debugging, and we wrote Python scripts to automate calls to GDB to extract function call stacks after crashes. For the key bytes, we use PIN [33] for taint analysis and judge whether tainted bytes come directly from POC bytes.

Trace Guided Fuzzing For the Key Bytes Guided Mutation strategy, we simply maintain the extracted key bytes as a dictionary and enter them as arguments when calling fuzzer. For the Nested Simulated Annealing algorithm, we save the function call sequences in the historical trace as several subscripts of a global array that records the test case coverage edges during Fuzzing. After executing a new coverage edge, we determine the state of the current test case by detecting whether the edge corresponds to the several subscripts of the global array or not. Additionally, we update the global state machine with this information. By doing so, we have implemented TrasnferFuzz with only negligible space and time complexity. This makes TransferFuzz no slower to execute than generic directed Fuzzing. Finally, we developed a shell script to automatically determine the completion of the verification process by checking the crash address via GDB.

V. EVALUATION

In this section, we evaluate TransferFuzz and the existing state-of-the-art (SOTA) works. We provide a comprehensive analysis, including an investigation into the bad cases of existing works, and identify the reasons for their shortcomings. Before that, we describe our experimental setup, the dataset used, and the compared works.

First, in the experiments, we aim to answer the following research questions:

RQ1: How TransferFuzz performs in the Vulnerability Verification of Reused Code task?

RQ2: How efficiently TransferFuzz triggers the vulnerability in target binaries?

RQ3: How does every component in TransferFuzz affect the overall performance?

Then, we detail the experimental setup. The system operates on Ubuntu 22.04, powered by an Intel Xeon CPU with 128 cores at 3.0GHz and hyperthreading capabilities. Each fuzzer is executed within a Docker container.

Finally, we employed the following metrics to evaluate the performance of different methods:

To answer RQ1, we use Precision and Recall to evaluate whether TransferFuzz can verify more vulnerabilities and reduce false positives compared to existing methods. Specifically, TP represents the number of correctly verified propagated vulnerabilities, while TN denotes the number of non-propagated vulnerabilities that are correctly detected as non-propagated. Conversely, FN indicates the number of non-propagated vulnerabilities that are incorrectly verified while FP presents the number of propagated vulnerabilities that are incorrectly detected as propagated. Moreover, the equations of Precision and Recall are as follows:

$$Precision = \frac{TP}{TP + FP} \tag{2}$$

$$Recall = \frac{TP}{TP + FN} \tag{3}$$

To answer the RQ2 and RQ3, we use the Time-to-Exposure (TTE) to evaluate the efficiency. Similar to previous work [23], each method was repeated 10 times to take the average μTTE with a time budget of 12 hours. We calculate the sum of μTTE for vulnerabilities successfully triggered 10 times and evaluate the percent TrasferFuzz speed up.

TTE: Time-to-Exposure (TTE) is used to measure the time used by the fuzzer to trigger the vulnerability.

A. Dataset

We performed code reuse detection on the datasets from AFLGo [20], SelectFuzz [31], and DAFL [25], extracting binaries with propagated vulnerable functions. Additionally, although OCTOPOCs does not publish its dataset, we found six of its PDF processing software on the internet. In total, we collected 15 CVE vulnerabilities that propagate due to code

TABLE II
ACCURACY IN THE VULNERABILITY VERIFICATION TASK

Model	TP	FP	TN	FN	Precision	Recall
LibAM	38	108	0	0	0.260	1.0
OCTOPOCs	15	0	108	23	1.0	0.395
AFLGo	27	0	108	11	1.0	0.711
Windranger	29	0	108	10	1.0	0.763
SelectFuzz	29	0	108	10	1.0	0.763
DAFL	20	0	108	18	1.0	0.526
TransferFuzz	**38**	**0**	**108**	**0**	**1.0**	**1.0**

Fig. 5. CPU and Memory Usage. The green bars represent CPU Usage, while the gray bars indicate Memory Usage (RES).

reuse and 76 related binaries. The dataset is summarized in Table 1.

Using LibAM, the state-of-the-art code reuse detection technology, we identified 146 new potential vulnerabilities by checking for the presence of vulnerable code. After manually filtering out 108 false positives, we found that the initial 15 CVE vulnerabilities had broader implications, affecting more software and extending to 53 distinct vulnerabilities. This dataset demonstrates the capability of TransferFuzz in vulnerability validation and proof-of-concept (POC) generation.

B. Compared Methods

In this evaluation, we introduced the comparison between TransferFuzz and various existing methods. The comparison methods are as follows:

LibAM [5]: We employed LibAM, the SOTA third-party library code reuse detection method, as a representative of code reuse-based approaches.

OCTOPOCs [24]: OCTOPOCs is the only one that has a similar idea to ours, employing symbolic execution and taint analysis for the verification of propagating vulnerable code.

AFLGo [20]: AFLGo is the earliest and classic Directed Fuzzing technique.

Windranger [23]: Windranger is a directed fuzzing technology that uses deviation basic blocks.

SelectFuzz [31]: SelectFuzz is an advanced directed fuzzing approach that employs selective instrumentation to enhance efficiency.

DAFL [25]: DAFL is an advanced directed fuzzing technology that uses data flow distance metrics

C. Answer to RQ 1: Accuracy of Verifying Propagated Vulnerability Code

In this section, we aim to assess whether TransferFuzz, in comparison with existing methodologies, can accurately validate vulnerabilities in propagated code.

The results presented in Table 2 indicate that TransferFuzz achieves 1.0 of precision and 1.0 of recall. Besides, TransferFuzz successfully validated all 38 propagated vulnerability samples. It can verify vulnerabilities more accurately and comprehensively than existing methods.

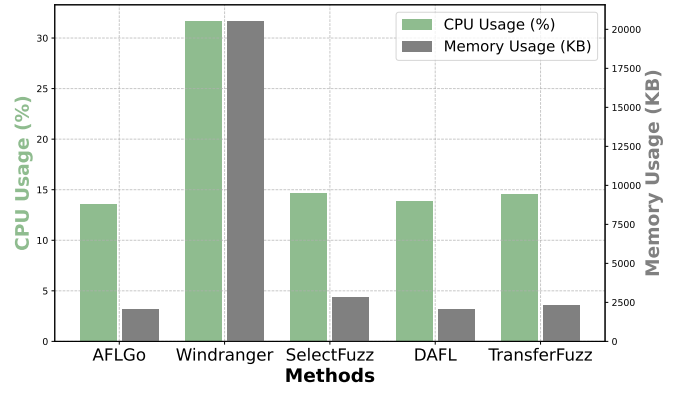LibAM is another method for achieving a recall of 1.0, as such reuse detection approaches assess code similarity without further validating whether vulnerabilities are triggerable. This results in a significantly low precision of 0.260.

OCTOPOCs is the first method to transfer Proof-of-Concepts (POCs) from basic binaries to target binaries. However, due to its susceptibility to path explosion and the requirement for target binaries to share the same input type as the basic binaries, it effectively only applies to the scenario where the POCs of source binaries and target binaries are consistent, which account for less than half of our datasets.

Other Directed Fuzzing methods demonstrate commendable precision. Yet, their recall rates are significantly lower than that of TransferFuzz. This disparity stems from their limited capacity to validate vulnerabilities with complex logic. Some vulnerabilities are difficult to trigger. TransferFuzz overcomes this challenge by utilizing historical traces derived from POCs in basic binaries.

> **Answering RQ1:** TransferFuzz demonstrates the capability to accurately validate vulnerabilities, achieving the highest precision and recall. TransferFuzz successfully validated 38 propagated vulnerability samples.

D. Answer to RQ 2: Efficiency in Triggering Vulnerabilities

In this section, we evaluate the speed of TransferFuzz compared to existing directed fuzzing methods. Note that we used empty files or simple test files (e.g., basic ELF or PDF files) from the target project as initial seeds for all methods, similar to SelectFuzz [31]. Although some POCs of basic binary can directly work on target binaries, they constitute only a small portion of our dataset (Several POCs from LibMing and OCTOPOCs). To ensure a fair evaluation of each fuzzer, we did not take shortcuts by directly using POCs from basic binaries, which is different from the settings in RQ1. We analyze each sample, elucidating the reasons why TransferFuzz outperforms existing methods.

According to Table 3, TransferFuzz ranks as the most rapid method, achieving a speed increase of 2.5 times over the state-of-the-art SelectFuzz. Almost all vulnerabilities are verified within 60 minutes, with over half of them confirmed within 2

TABLE III
THE NUMBER OF RUNS AND EFFICIENCY FOR EACH VULNERABILITY.

No.	Vulnerability	AFLGo		Windranger		SelectFuzz		DAFL		TransferFuzz	
		Runs	μTTE	Runs	μTTE	Runs	μTTE	Runs	μTTE	Runs	μTTE
1	thumbnail-2016-10095	10	10m35s	10	3m27s	10	2m13s	10	3m38s	10	**0m56s**
2	tiffcmp-2016-10095	10	8m24s	10	5m16s	10	2m32s	10	4m17s	10	**1m49s**
3	tiffsplit-2016-5318	10	7m27	10	6m5s	10	1m24s	10	2m53s	10	**1m28s**
4	tiffcmp-2016-5318	10	7m12s	10	4m39s	10	3m5s	10	4m24s	10	**1m38s**
5	objdump-2016-4487	10	27m23s	10	5m32s	10	14m17s	10	9m54s	10	**0m5s**
6	nm-new-2016-4487	10	35m31s	10	6m12s	10	12m32s	10	13m56s	10	**0m40s**
7	addr2line-2016-4487	10	1h5m45s	10	1h14m23s	10	43m23s	10	1h3m24s	10	**0m38s**
8	objdump-2016-4489	10	48m5s	10	7m16s	10	17m23s	10	13m45s	10	**0m4s**
9	nm-new-2016-4489	10	34m51s	10	3m42s	10	13m51s	10	9m13s	10	**0m27s**
10	addr2line-2016-4489	10	1h47m15s	10	1h52m0s	10	1h3m25s	10	1h24m26s	10	**0m51s**
11	objdump-2016-4490	10	21m45s	10	4m14s	10	13m26s	10	9m42s	10	**0m4s**
12	nm-new-2016-4490	10	35m23s	10	11m24s	10	9m52s	10	13m26s	10	**1m3s**
13	addr2line-2016-4490	10	2h4m21s	10	1h3m25s	10	58m25s	10	42m52s	10	**0m53s**
14	objdump-2016-4491	0	N.A.	0	N.A.	0	N.A.	0	N.A.	10	**3m37s**
15	nm-new-2016-4491	0	N.A.	0	N.A.	0	N.A.	0	N.A.	10	**4m15s**
16	addr2line-2016-4491	0	N.A.	0	N.A.	0	N.A.	0	N.A.	10	**7m14s**
17	objdump-2016-4492	10	56m24s	10	46m11s	10	23m15s	10	35m1s	10	**0m32s**
18	nm-new-2016-4492	10	1h2m42s	10	58m15s	10	44m35s	10	39m25s	10	**0m40s**
19	addr2line-2016-4492	0	N.A.	0	N.A.	3	2h46m	2	3h41m58s	10	**0m17s**
20	objdump-2016-6131	0	N.A.	0	N.A.	0	N.A.	0	N.A.	10	**0m4s**
21	nm-new-2016-6131	0	N.A.	0	N.A.	0	N.A.	0	N.A.	10	**0m56s**
22	addr2line-2016-6131	0	N.A.	0	N.A.	0	N.A.	0	N.A.	10	**0m32s**
23	objcopy-2017-7303	0	N.A.	3	2h33m	5	1h57m12s	0	N.A.	6	**1h30m2s**
24	swftophp-2017-11733	10	5m16s	10	1m59s	10	0m32s	10	1m23s	10	**0m6s**
25	swftoperl-2017-11733	10	4m25s	10	1m42s	10	**0m34s**	10	2m0s	10	0m52s
26	swftopython-2017-11733	10	6m43s	10	3m1s	10	0m46s	10	1m25s	10	**0m28s**
27	swftotcl-2017-11733	10	3m25s	10	2m48s	10	1m22s	10	2m34s	10	**0m25s**
28	swftocxx-2018-8807	10	2h24m17s	10	40m45s	10	32m14s	0	N.A.	10	**20m2s**
29	swftoperl-2018-8807	10	2h4m17s	10	51m32s	10	19m15s	0	N.A.	10	**9m21s**
30	swftopython-2018-8807	10	2h37m42s	10	47m13s	10	26m41s	0	N.A.	10	**17m2s**
31	swftotcl-2018-8807	10	2h8m32s	10	54m21s	10	29m1s	0	N.A.	10	**22m49s**
32	swftocxx-2018-8962	10	3h43m54s	10	1h13m45s	10	45m15s	0	N.A.	10	**33m42s**
33	swftoperl-2018-8962	10	3h24m16s	10	56m34s	10	33m25s	0	N.A.	10	**29m15s**
34	swftopython-2018-8962	10	3h52m1s	10	1h3m52s	10	58m21s	0	N.A.	10	**55m20s**
35	swftotcl-2018-8962	10	3h35m15s	10	52m16s	10	1h1m15s	0	N.A.	10	**46m16s**
μTTE inc (all binaries)		**+850%**		**+344%**		**+248%**		**+2620%**			

minutes. Notably, without using POCs of basic binary, none of the methods, including TransferFuzz, could generate POCs for three vulnerabilities (CVE-2017-18267, CVE-2018-11102, and CVE-2018-20330). This limitation arises because existing methods are prototypes and require additional engineering efforts, such as in-process instrumentation, to adapt to more software. We plan to expand existing fuzzers and undertake these engineering efforts in the future.

Other methods sometimes take several hours to trigger vulnerabilities. For some complex vulnerabilities, such as CVE-2016-4491, CVE-2016-4492, and CVE-2016-6131, they failed to trigger them. Owing to the KBGM algorithm, TransferFuzz can shorten the time cost for triggering from several hours to a few minutes (e.g., Vul-10). Even for vulnerabilities without complex branches, TransferFuzz significantly speeds up existing methods (e.g., Vul-28 to Vul-35) using the NSA module.

Further explanation of the results is provided in Section V-F.

We also recorded the resource consumption of these methods, measuring the maximum CPU usage and memory usage (RES) of TransferFuzz and existing tools during the fuzzing process of each vulnerability. As shown in Figure 5, the resource consumption of TransferFuzz is comparable to AFLGo, SelectFuzz, and DAFL. However, Windranger's memory consumption increased significantly due to the continuous use of taint analysis during the fuzzing process. Despite this, the resource consumption remains acceptable and does not affect the applicability of vulnerability verification.

> **Answering RQ2:** TransferFuzz consistently outperforms existing methods in terms of both speed and the number of vulnerabilities verified.

E. Answer to RQ 3: Impact of Different Components

In this section, we aim to validate the effectiveness of each module within TransferFuzz. We conducted ablation experiments by removing the Key Bytes Guided Mutation strategy, denoted as No KBGM, and the Nested Simulated Annealing algorithm, denoted as No NSA.

In the case of No KBGM, TransferFuzz experienced a notable decline in efficiency due to the lack of key-bytes traces from taint analysis, hindering its ability to bypass essential constraints. This reduction in performance was particularly pronounced for vulnerabilities like CVE-2016-4490, CVE-2016-4491, CVE-2016-4492, and CVE-2016-6131. In contrast, the impact on CVE-2017-11733 was minimal, illustrating that KBGM's effectiveness in extracting key bytes is not universal. Despite these challenges, No KBGM still outperformed the alternative methods in speed.

In the case of No NSA, TransferFuzz experienced a performance decline, although not as pronounced as observed with No KBGM. Despite this, a steady reduction in efficiency was noted since the NSA contributed to enhancements across all examined vulnerabilities.

> **Answering RQ3:** KBGM and NSA both contribute to the performance of TransferFuzz. KBGM can significantly enhance performance under specific circumstances, while NSA consistently improves performance in all cases.

F. Result Analysis

In this section, we analyzed the spread of these vulnerabilities and tried to explain the performance of TransferFuzz in these cases.

Vul-1 to Vul-4. Due to different processing methods for tiff files, the POCs between binaries such as thumbnail and tiffsplit are not adaptable (cann't use basic POC directly in target binaries). However, they all reuse the same vulnerable function and do not perform security checks on the key parameters of the vulnerability, leading to the spread of the vulnerability between these binaries.

These are simple vulnerabilities that all methods can trigger within 10 minutes. Among them, SelectFuzz and TransferFuzz perform the best due to their use of selective instrumentation.

Vul-5 to Vul-23. These vulnerabilities successfully spread across the *cxxfilt*, *objdump*, *nm-new*, and *addr2line*. However, *cxxfilt* takes a string as input, which is directly used as the parameter of the vulnerable function. In contrast, the other binaries require an ELF file as input. Therefore, it is necessary to construct appropriate input to trigger the vulnerability in target binaries instead of directly using basic POC.

By extracting the key bytes in the POC of *cxxfilt* through KBGM, TransferFuzz only needs to insert these key bytes into specific locations of the initial seed (a simple ELF file). Consequently, whether dealing with complex vulnerabilities (such as CVE-2016-4491 and CVE-2016-4492) or simpler

TABLE IV
EFFICIENCY OF DIFFERENT MODULES OF TRANSFERFUZZ.

CVE-ID	Targets	No KBGM	No NSA	TransferFuzz
2016-10095	thumbnail	1m59s	1m2s	**0m56s**
	tiffcmp	3m1s	2m24s	**1m49s**
2016-5318	tiffsplit	1m54s	1m31s	**1m28s**
	tiffcmp	2m6s	1m56s	**1m38s**
2016-4487	objdump	6m32s	0m9s	**0m5s**
	nm-new	7m45s	0m51s	**0m40s**
	addr2line	26m51s	0m56s	**0m38s**
2016-4489	objdump	14m2s	0m58s	**0m54s**
	nm-new	11m41s	0m34s	**0m27s**
	addr2line	47m26s	1m14s	**0m51s**
2016-4490	objdump	12m51s	0m4s	**0m4s**
	nm-new	9m3s	1m18s	**1m3s**
	addr2line	34m21s	1m21s	**0m53s**
2016-4491	objdump	N.A.	6m19s	**3m37s**
	nm-new	N.A.	6m43s	**4m15s**
	addr2line	N.A.	9m12s	**7m14s**
2016-4492	objdump	19m51s	**0m29s**	0m32s
	nm-new	25m32s	0m47s	**0m40s**
	addr2line	1h18m3s(5)	0m36s	**0m17s**
2016-6131	objdump	N.A.	0m5s	**0m4s**
	nm-new	N.A.	1m6s	**0m56s**
	addr2line	N.A.	0m47s	**0m32s**
2017-7303	objcopy	1h43m16s(7)	2h14m(4)	**1h30m(6)**
2017-11733	swftophp	18s	0m26s	**0m6s**
	swftoperl	0m58s	0m43s	**0m52s**
	swftopython	0m41s	0m35s	**0m28s**
	swftotcl	1m12s	0m37s	**0m25s**
2018-8807	swftocxx	**18m49s**	28m13s	20m2s
	swftoperl	9m52s	16m4s	**9m21s**
	swftopython	18m34s	21m15s	**17m2s**
	swftotcl	22m56s	23m35s	**22m49s**
2018-8962	swftocxx	37m25s	41m38s	**33m42s**
	swftoperl	31m25s	34m21s	**29m15s**
	swftopython	56m16	57m54s	**55m20s**
	swftotcl	51m46s	**43m19s**	46m16s

ones (e.g., CVE-2016-4487), TransferFuzz can indiscriminately insert key bytes into random locations in the ELF file to quickly trigger the vulnerabilities. In contrast, other methods need to constantly mutate bytes to meet different conditions and often fail to trigger complex vulnerabilities.

Vul-24 to Vul-35. For the vulnerabilities in LibMing, although their POCs are universal, we did not use the original POCs during fuzzing to test the capabilities of different fuzzers. Owing to selective instrumentation, both SelectFuzz and TrasnferFuzz triggered these vulnerabilities in a shorter time. Due to the improvement of NSA, TrasnferFuzz is slightly better than SelectFuzz. Unfortunately, DAFL failed to verify CVE-2018-8807 and CVE-2018-8962, which is consistent with the experiments reported in the DAFL paper [25].

VI. THREATS TO VALIDITY

The first concern is the effectiveness of TransferFuzz. First, not all vulnerabilities allow for the extraction of key-bytes traces, such as CVE-2017-7303. Despite this, many vulnerabilities do produce such traces, as evidenced by the Binutils dataset where all binaries have key-byte traces, significantly enhancing the performance of our experiments. In addition, some binaries may slow down the fuzzing process due to complex branch constraints or missing key-bytes traces. Nonetheless, TransferFuzz consistently completed verifications within two hours, with over half of the cases resolved in under 1

minutes, which shows the effectiveness of TransferFuzz in verifying vulnerability code propagation.

The second concern is the obstacles to implementation. All processes of TransferFuzz are automated, with only minimal manual work required to write the fuzz driver before fuzzing the target binary, which is a common issue with fuzzing techniques. Although recent works propose automating this task [34], these efforts are considered orthogonal to ours. This remaining manual work does not significantly affect the scalability and is much more straightforward for security researchers and developers than manual verification vulnerabilities. Even users familiar with fuzzing technology or the target software can use our tool to verify the target software.

The third concern is the evaluation dataset. The dataset for evaluation was selected not to intentionally showcase TransferFuzz's strengths but to maintain objectivity. It encompasses all instances from AFLGo and SelectFuzz related to vulnerability code propagation, alongside as many cases as possible from OCTOPOCs (their dataset not being available). Our objective was to employ unbiased datasets for existing methods. However, further empirical studies across a wider range of projects are essential to fully generalize our evaluation results.

VII. RELATED WORKS

A. Code Reuse Detection

A large number of works have been proposed for code reuse detection. [35], [36]. First, some work uses constant features to detect code reuse. they identify TPL reuse within the target binary by extracting identical constant features, such as strings [37], [38], function names [8], and jump tables [9], from both the target binary and the basic binary. In addition, some researchers use binary code similarity detection technology to detect reused code. they involve comparing all the functions of the target binary with those of the basic binary. Subsequently, a predetermined threshold is set to establish whether reuse has occurred. Gitz [39] and VIVA [40] use text hashing to represent the function features. BinSim [41] and Bingo [42] use Symbolic execution-based techniques to offer a more robust approach to function similarity detection. Besides, more works [11], [12] use deep learning technology to learn the semantics of functions.

All of these techniques can be used as a precursor to TransferFuzz for detecting reused vulnerable code. However they do not further validate these vulnerabilities, and using TransferFuzz can further filter out vulnerability false positives in code reuse detection results.

B. Vulnerability Detection

Vulnerability detection has long been a popular and significant field in computer science and cybersecurity. Researchers aim to detect vulnerabilities in newly developed code by extracting features from vulnerable functions and determining if these vulnerable functions exist within the target code.

Early works in this field, such as Bingo [42] and SAFE [43], calculated function similarity by directly comparing the vulnerable function with all functions present in the target code. More recent approaches, including MVP [44] and VIVA [40], detect vulnerabilities by employing data stream slicing techniques to identify the presence of vulnerability code and the absence of patch code. Additionally, Fiber [16] and PDiff [17] attempt to extract deep patch code semantics by utilizing symbolic execution to ascertain whether the target vulnerable function has been patched or not.

These vulnerability detection efforts likewise do not confirm whether the vulnerability is triggerable. All they do is detect whether a patch exists. Even if the patch doesn't exist, the vulnerability is not necessarily triggerable in a new context.TransferFuzz fills this gap nicely.

C. Directed Grey-box Fuzzing

Fuzzing, an essential technique in vulnerability discovery for real-world software like parsers [45], network protocols [46], [47], web browsers [48], [49], mobile apps [50], and OS kernels [51]–[53], has seen significant advancements, particularly in Grey-box fuzzing [54]–[56].

Directed Grey-box Fuzzing (DGF) addresses the persistent challenge of generating test cases to expose specific program bugs. AFLGo [20], one of the earliest directed fuzzers, prioritizes test case generation by favoring nodes closer to the target node within the Control-Flow Graph (CFG). Hawkeye [57] and WindRanger [23] build upon the AFLGo approach, enhancing it to provide more informative feedback for precise distance computation.

TransferFuzz is the first to use historical fuzzing data across software to guide new fuzz, which greatly improves the capabilities of the Directed Fuzzer.

VIII. CONCLUSION

Code reuse in software development commonly leads to the propagation of vulnerability code. Existing methods primarily focus on detecting the presence of vulnerable code in target software, without automatically verifying whether the vulnerability poses a real threat in the new context. This deficiency results in many false positives. In this paper, we introduce TransferFuzz, a novel vulnerability verification framework, to verify whether vulnerabilities propagated through code reuse are actually vulnerable in new software. Our experiments show that TransferFuzz rapidly detects the exact software scope affected by the vulnerability and generates Proofs-of-Concept to provide strong evidence for the propagation of the vulnerability. Furthermore, TransferFuzz expands the impacted software scope of ten vulnerabilities in the CVE report from 15 binaries to 53, underscoring its effectiveness in verifying propagated vulnerability code.

IX. ACKNOWLEDGE

REFERENCES

[1] "github." 2023, https://github.com/.
[2] "Sourceforge." 2023, https://sourceforge.net/.
[3] "Vcpkg." 2023, https://vcpkg.io/.
[4] Synopsys, "Synopsys 2022 open source security and risk analysis report." 2022, https://www.synopsys.com/software-integrity/resources/analyst-reports/open-source-security-risk-analysis.html.
[5] S. Li, Y. Wang, C. Dong, S. Yang, H. Li, H. Sun, Z. Lang, Z. Chen, W. Wang, H. Zhu *et al.*, "Libam: An area matching framework for detecting third-party libraries in binaries," *arXiv preprint arXiv:2305.04026*, 2023.
[6] "Cve," 2023, https://cve.mitre.org/.
[7] "Nvd," 2023, https://nvd.nist.gov/.
[8] R. Duan, A. Bijlani, M. Xu, T. Kim, and W. Lee, "Identifying open-source license violation and 1-day security risk at large scale," in *Proceedings of the 2017 ACM SIGSAC Conference on computer and communications security*, 2017, pp. 2169–2185.
[9] Z. Yuan, M. Feng, F. Li, G. Ban, Y. Xiao, S. Wang, Q. Tang, H. Su, C. Yu, J. Xu *et al.*, "B2sfinder: detecting open-source software reuse in cots software," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1038–1049.
[10] B. Zhao, S. Ji, J. Xu, Y. Tian, Q. Wei, Q. Wang, C. Lyu, X. Zhang, C. Lin, J. Wu *et al.*, "A large-scale empirical analysis of the vulnerabilities introduced by third-party components in iot firmware," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 442–454.
[11] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 363–376.
[12] H. Wang, W. Qu, G. Katz, W. Zhu, Z. Gao, H. Qiu, J. Zhuge, and C. Zhang, "Jtrans: Jump-aware transformer for binary code similarity detection," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 1–13.
[13] S. Yang, L. Cheng, Y. Zeng, Z. Lang, H. Zhu, and Z. Shi, "Asteria: Deep learning-based ast-encoding for cross-platform binary code similarity detection," in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2021, pp. 224–236.
[14] S. Li, C. Dong, Y. Wang, W. Liu, W. Wang, H. Li, H. Zhu, and L. Sun, "Libdi: A direction identification framework for detecting complex reuse relationships in binaries," in *MILCOM 2023-2023 IEEE Military Communications Conference (MILCOM)*. IEEE, 2023, pp. 741–746.
[15] L. Jiang, H. Yuan, Q. Tang, S. Nie, S. Wu, and Y. Zhang, "Third-party library dependency for large-scale sca in the c/c++ ecosystem: How far are we?" in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2023. New York, NY, USA: Association for Computing Machinery, 2023, pp. 1383–1395. [Online]. Available: https://doi.org/10.1145/3597926.3598143
[16] H. Zhang and Z. Qian, "Precise and accurate patch presence test for binaries," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 887–902.
[17] Z. Jiang, Y. Zhang, J. Xu, Q. Wen, Z. Wang, X. Zhang, X. Xing, M. Yang, and Z. Yang, "Pdiff: Semantic-based patch presence testing for downstream kernels," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 1149–1163.
[18] C. Dong, S. Li, S. Yang, Y. Xiao, Y. Wang, H. Li, Z. Li, and L. Sun, "Libvdiff: Library version difference guided oss version identification in binaries," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3623336
[19] S. Yang, Z. Xu, Y. Xiao, Z. Lang, W. Tang, Y. Liu, Z. Shi, H. Li, and L. Sun, "Towards practical binary code similarity detection: Vulnerability verification via patch semantic analysis," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 6, sep 2023. [Online]. Available: https://doi.org/10.1145/3604608
[20] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, 2017, pp. 2329–2344.

[21] M.-D. Nguyen, S. Bardin, R. Bonichon, R. Groz, and M. Lemerre, "Binary-level directed fuzzing for {Use-After-Free} vulnerabilities," in *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, 2020, pp. 47–62.
[22] H. Huang, Y. Guo, Q. Shi, P. Yao, R. Wu, and C. Zhang, "Beacon: Directed grey-box fuzzing with provable path pruning," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 36–50.
[23] Z. Du, Y. Li, Y. Liu, and B. Mao, "Windranger: a directed greybox fuzzer driven by deviation basic blocks," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2440–2451.
[24] S. Kwon, S. Woo, G. Seong, and H. Lee, "Octopocs: automatic verification of propagated vulnerable code using reformed proofs of concept," in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2021, pp. 174–185.
[25] T. E. Kim, J. Choi, K. Heo, and S. K. Cha, "{DAFL}: Directed grey-box fuzzing guided by data dependency," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 4931–4948.
[26] P. Zong, T. Lv, D. Wang, Z. Deng, R. Liang, and K. Chen, "{FuzzGuard}: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning," in *29th USENIX security symposium (USENIX security 20)*, 2020, pp. 2255–2269.
[27] J. Dai, Y. Zhang, H. Xu, H. Lyu, Z. Wu, X. Xing, and M. Yang, "Facilitating vulnerability assessment through poc migration," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 3300–3317. [Online]. Available: https://doi.org/10.1145/3460120.3484594
[28] S. Kirkpatrick, C. D. Gelatt Jr, and M. P. Vecchi, "Optimization by simulated annealing," *science*, vol. 220, no. 4598, pp. 671–680, 1983.
[29] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "{AFL++}: Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.
[30] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "Redqueen: Fuzzing with input-to-state correspondence." in *NDSS*, vol. 19, 2019, pp. 1–15.
[31] C. Luo, W. Meng, and P. Li, "Selectfuzz: Efficient directed fuzzing with selective path exploration," in *2023 IEEE Symposium on Security and Privacy (SP)*, 2023, pp. 2693–2707.
[32] "Gdb." 2023, https://www.gnu.org/software/gdb/.
[33] "Pin." 2023, https://software.intel.com/sites/landingpage/pintool/docs/98484/Pin/html/index.html.
[34] C. Zhang, Y. Li, H. Zhou, X. Zhang, Y. Zheng, X. Zhan, X. Xie, X. Luo, X. Li, Y. Liu *et al.*, "Automata-guided control-flow-sensitive fuzz driver generation." in *USENIX Security Symposium*, 2023, pp. 2867–2884.
[35] S. Alrabaee, M. Debbabi, and L. Wang, "A survey of binary code fingerprinting approaches: Taxonomy, methodologies, and features," *ACM Computing Surveys (CSUR)*, vol. 55, no. 1, pp. 1–41, 2022.
[36] I. U. Haq and J. Caballero, "A survey of binary code similarity," *ACM Computing Surveys (CSUR)*, vol. 54, no. 3, pp. 1–38, 2021.
[37] A. Hemel, K. T. Kalleberg, R. Vermaas, and E. Dolstra, "Finding software license violations through binary code clone detection," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, 2011, pp. 63–72.
[38] W. Tang, P. Luo, J. Fu, and D. Zhang, "Libdx: A cross-platform and accurate system to detect third-party libraries in binary code," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 104–115.
[39] Y. David, N. Partush, and E. Yahav, "Similarity of binaries through re-optimization," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017, pp. 79–94.
[40] Y. Xiao, Z. Xu, W. Zhang, C. Yu, L. Liu, W. Zou, Z. Yuan, Y. Liu, A. Piao, and W. Huo, "Viva: Binary level vulnerability identification via partial signature," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2021, pp. 213–224.
[41] J. Ming, D. Xu, Y. Jiang, and D. Wu, "{BinSim}: Trace-based semantic binary diffing via system call sliced segment equivalence checking," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 253–270.
[42] M. Chandramohan, Y. Xue, Z. Xu, Y. Liu, C. Y. Cho, and H. B. K. Tan, "Bingo: Cross-architecture cross-os binary search," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 678–689.

[43] L. Massarelli, G. A. D. Luna, F. Petroni, R. Baldoni, and L. Querzoni, "Safe: Self-attentive function embeddings for binary similarity," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2019, pp. 309–329.

[44] Y. Xiao, B. Chen, C. Yu, Z. Xu, Z. Yuan, F. Li, B. Liu, Y. Liu, W. Huo, W. Zou *et al.*, "{MVP}: Detecting vulnerabilities using {Patch-Enhanced} vulnerability signatures," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1165–1182.

[45] B. Mathis, R. Gopinath, M. Mera, A. Kampmann, M. Höschele, and A. Zeller, "Parser-directed fuzzing," in *Proceedings of the 40th acm sigplan conference on programming language design and implementation*, 2019, pp. 548–560.

[46] H. Gascon, C. Wressnegger, F. Yamaguchi, D. Arp, and K. Rieck, "Pulsar: Stateful black-box fuzzing of proprietary network protocols," in *Security and Privacy in Communication Networks: 11th EAI International Conference, SecureComm 2015, Dallas, TX, USA, October 26-29, 2015, Proceedings 11*. Springer, 2015, pp. 330–347.

[47] D. Liu, V.-T. Pham, G. Ernst, T. Murray, and B. I. Rubinstein, "State selection algorithms and their impact on the performance of stateful network protocol fuzzing," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022, pp. 720–730.

[48] H. Han, D. Oh, and S. K. Cha, "Codealchemist: Semantics-aware code generation to find vulnerabilities in javascript engines." in *NDSS*, 2019.

[49] J. Wang, B. Chen, L. Wei, and Y. Liu, "Superion: Grammar-aware greybox fuzzing," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 724–735.

[50] L. Luo, Q. Zeng, B. Yang, F. Zuo, and J. Wang, "Westworld: Fuzzing-assisted remote dynamic symbolic execution of smart apps on iot cloud platforms," in *Annual Computer Security Applications Conference*, 2021, pp. 982–995.

[51] J. Choi, K. Kim, D. Lee, and S. K. Cha, "Ntfuzz: Enabling type-aware kernel fuzzing on windows with static binary analysis," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 677–693.

[52] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin, "Razzer: Finding kernel race bugs through fuzzing," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 754–768.

[53] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "{kAFL}:{Hardware-Assisted} feedback fuzzing for {OS} kernels," in *26th USENIX security symposium (USENIX Security 17)*, 2017, pp. 167–182.

[54] M. Böhme, V. J. Manès, and S. K. Cha, "Boosting fuzzer efficiency: An information theoretic perspective," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 678–689.

[55] A. Fioraldi, D. C. D'Elia, and E. Coppa, "Weizz: Automatic grey-box fuzzing for structured binary formats," in *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, 2020, pp. 1–13.

[56] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2312–2331, 2019.

[57] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu, "Hawkeye: Towards a desired directed grey-box fuzzer," in *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, 2018, pp. 2095–2108.