ChainFuzz: Exploiting Upstream Vulnerabilities in Open-Source Supply Chains

Peng Deng, Lei Zhang, Yuchuan Meng, Zhemin Yang, Yuan Zhang, and Min Yang
Fudan University, China
{pdeng21, ycmeng22}@m.fudan.edu.cn, {zxl, yangzhemin, yuanxzhang, m_yang}@fudan.edu.cn

Abstract

Software supply chain attacks pose an increasingly severe threat to the security of downstream software worldwide. A common method to mitigate these risks is Software Composition Analysis (SCA), which helps developers identify vulnerable dependencies. However, studies show that popular SCA approaches often suffer from high false positive rates. As a result, developers spend significant time manually validating these alerts, which delays the detection and remediation of genuinely exploitable upstream vulnerabilities.

In this paper, we propose ChainFuzz, an automated approach for validating upstream vulnerabilities in downstream software by generating Proof-of-Concepts (PoCs). To achieve this, ChainFuzz addresses three key challenges. First, intra-layer code and constraints. Downstream software introduces custom code and sanity checks that significantly alter the triggering paths and conditions of upstream vulnerabilities. Second, inter-layer dependencies. Software supply chains often involve cross-layer control-flow and data-flow dependencies between conditional statements across different layers. Third, long supply chains. Transitive dependencies in long chains result in intricate exploitation paths, making it challenging to explore large code spaces and handle deeply nested constraints effectively.

We comprehensively evaluate ChainFuzz using our dataset, which comprises 66 unique vulnerability and supply chain combinations. Our results demonstrate its effectiveness and practicality in generating PoCs for both direct and transitive vulnerable dependencies. Additionally, we compare ChainFuzz with representative fuzzing tools: AFLGo, AFL++, and NestFuzz, highlighting its superior performance in downstream PoC generation.

1 Introduction

Modern software development heavily relies on open-source third-party libraries to enhance efficiency and security [9, 34, 44]. These libraries act as critical building blocks in software supply chains, significantly impacting both the development and security of downstream software. However, vulnerabilities in upstream libraries can propagate throughout the supply chain, posing risks to downstream systems and enabling supply chain attacks [3, 41]. For instance, the CVE-2024-3094 [30] vulnerability (with a CVSS score of 10) in the `xz/liblzma` library allows malicious attackers to gain remote access to downstream systems that depend on it.

Several approaches have been proposed to identify vulnerabilities inherited from open-source libraries in downstream software [13, 17, 32, 36, 41]. Unfortunately, these methods often suffer from high false positive rates [13], rendering them impractical and inefficient in real-world scenarios. For example, OWASP DC [32], a well-known tool, employs Software Composition Analysis (SCA) to identify and flag dependent libraries with known vulnerabilities. However, since it does not verify whether these vulnerabilities can actually be triggered or exploited in downstream software, Ponta *et al.* [47] reveal that 88.8% of the vulnerabilities reported by OWASP DC are false positives. Other techniques [13, 17, 36] have attempted to assess the *reachability* of upstream vulnerabilities, such as analyzing call graphs to determine whether the vulnerable functions are invoked. Nevertheless, these methods also produce false alerts because they fail to verify whether necessary conditions for exploitation are satisfied.

High rates of false alerts for upstream vulnerabilities delay the resolution of real, exploitable vulnerabilities, allowing them to persist in software [9, 35]. A common strategy to mitigate supply chain attacks is updating vulnerable upstream libraries. However, this process is often risky and time-consuming [25, 27]. Adapting to new interfaces typically requires substantial code changes, and updates can introduce new vulnerabilities that necessitate recertification and additional modifications. Without clear evidence of an upstream vulnerability's exploitability, developers must spend considerable time on manual validation, making updates to individual components costly and inefficient. Studies also show that downstream developers frequently avoid upgrading third-party libraries to prevent breaking changes in their projects [11, 18].

We present ChainFuzz, an automated approach for verifying the *exploitability* of upstream vulnerabilities by generating Proof-of-Concepts (PoCs) for downstream software. This approach enables downstream developers to assess the impact of upstream vulnerabilities quickly and encourages them to implement necessary mitigation measures, such as promptly updating dependency versions. However, achieving this is far from straightforward.

First, while vulnerability reports may include PoCs to reproduce vulnerability-related failures [7], and various PoC generation techniques have been proposed [2, 15, 16, 20, 48], e.g., directed grey-box fuzzing [2, 15, 16], these PoCs are often tailored for upstream software and cannot be directly applied to downstream software. Our study (elaborated in §4) reveals that only 3.25% of upstream PoCs can be directly used to trigger the same vulnerabilities in downstream software.

Second, adapting upstream PoCs for downstream software is challenging because it requires a deep understanding of program dependencies and code implementations. Downstream software often introduces customized code and additional sanity checks, which significantly alter the triggering paths and conditions of vulnerabilities, rendering the original PoCs ineffective. Moreover, there are *control-flow* and *data-flow* dependencies between the conditional statements across different layers. However, existing approaches [5, 6] mainly focus on dependencies within individual projects, making them ineffective for handling complex cross-layer dependencies.

Third, while testing both upstream libraries and downstream software as a unified entity might seem feasible, it presents significant challenges and is largely impractical. For example, constructing a cross-software *control-flow graph* is difficult and imprecise [2], making it challenging to manage the extensive dependencies present in modern software. Modern software frequently incorporates numerous *transitive* dependencies [21, 34], where vulnerabilities can propagate through the supply chain and be exploited in downstream software. In particular, if software S_2 depends on S_1, and S_1 depends on S_0, S_2 has a direct dependency on S_1 and a transitive dependency on S_0. These transitive dependencies extend the length of software supply chains, creating increasingly complex and concealed exploitation paths.

To address these challenges, ChainFuzz employs two key techniques: *cross-layer differential directed fuzzing* and *bottom-up proof-of-concept generation*. The core insight behind ChainFuzz is that the original PoC for an upstream vulnerability contains critical execution context and bug-triggering conditions, which can be utilized to guide the generation of PoCs for downstream software. Specifically, the original and downstream PoCs should produce similar execution traces within the upstream library.

Specifically, the guidance comes from two perspectives. First, to generate PoCs for direct vulnerable dependencies, ChainFuzz identifies *waypoints* between upstream and downstream software and targets them for directed fuzzing. ChainFuzz divides each input's execution trace into the downstream trace T_d and the upstream trace T_u. During exploration, it prioritizes the *diversity* of T_d and the *exploitability* of T_u. In the exploitation phase, it applies trace differential guided mutation strategies to finely adjust T_u, making it as similar as possible to the vulnerable execution trace T_v of the original PoC. Second, to address the path explosion issue common in traditional top-down approaches, ChainFuzz introduces a bottom-up strategy to generate PoCs for transitive vulnerable dependencies in long supply chains. For example, if S_0 contains a vulnerability, ChainFuzz first generates PoCs for S_1, and then uses these PoCs to guide the generation of PoCs for S_2. To enhance this process, ChainFuzz incorporates *PoC prioritization* and *task revisiting* strategies to select the most promising PoC at each dependency layer for guidance and to handle nested cross-layer dependencies in the supply chain. Further details are provided in §3.

We implement a prototype of ChainFuzz and evaluate its effectiveness in generating PoCs for upstream vulnerabilities. Our evaluation is based on a ground truth dataset consisting of 21 real-world vulnerabilities and 66 unique $\langle$vulnerability, supply-chain$\rangle$ pairs. We compare ChainFuzz with AFLGo-Up, AFLGo-Down, NestFuzz, and AFL++, demonstrating that it outperforms these baselines in both efficiency and effectiveness when generating downstream PoCs. Furthermore, ChainFuzz found eight zero-day vulnerabilities in specific downstream software due to their dependency on the vulnerable upstream components.

Compared to existing works, ChainFuzz stands out for its ability to generate PoCs for downstream software, enabling precise validation of the exploitability of upstream vulnerabilities. Additionally, we discovered that simply updating the vulnerable dependency to a patched version—a common mitigation strategy—may not always be effective. For example, using the PoC generated by ChainFuzz, downstream software may trigger a different upstream vulnerability or even a new vulnerability after the dependency is updated to a patched version. We believe these findings provide valuable insights for developers and security researchers.

In summary, we make the following contributions:

- We propose ChainFuzz, an effective approach that can validate the exploitability of upstream vulnerabilities in the software supply chain by generating PoCs.
- ChainFuzz conducts cross-layer differential directed fuzzing to explore the code space in the downstream software and handle cross-layer dependencies.
- ChainFuzz employs bottom-up PoC generation to generate PoCs for transitive dependencies in long supply chains effectively.
- We conduct comprehensive experiments to assess the effectiveness and practicality of ChainFuzz, significantly outperforming existing approaches.

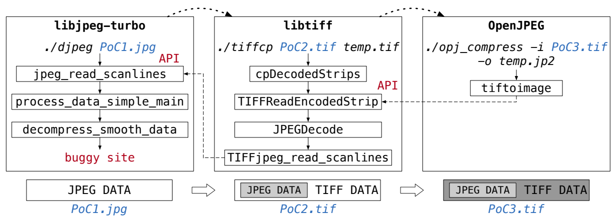


Figure 1: The propagation of vulnerability CVE-2021-29390.

2 Motivation

In this section, we first present a real-world example to illustrate the propagation of a vulnerability and highlight the associated research challenges (§2.1). We then introduce our proposed solutions (§2.2).

2.1 Motivating Example

Figure 1 illustrates the propagation of the vulnerability CVE-2021-29390 [1]. This is a high-risk *Out-of-Bounds Write* vulnerability in the widely used open-source library *libjpeg-turbo*, exploitable through a malformed JPEG file, denoted as PoC1.jpg in Figure 1. Furthermore, because *libjpeg-turbo* is a dependency of *libtiff*, which is in turn transitively depended upon by *OpenJPEG*, the vulnerability can potentially be triggered in these two downstream libraries, as demonstrated by PoC2.tif and PoC3.tif, respectively. In the following, we detail why and how the vulnerability in *libjpeg-turbo* propagates to *libtiff* and *OpenJPEG*.

Firstly, consider *libjpeg-turbo→libtiff*. *libtiff* is a popular software for handling TIFF files and relies on *libjpeg-turbo* to process embedded JPEG data within these files. During the processing of the TIFF file PoC2.tif, *libtiff* initially employs custom code to handle the outer layer of TIFF data. When encountering embedded JPEG data, *libtiff* transfers it to *libjpeg-turbo* through the API function *jpeg_read_scanlines()*. This API function eventually calls the vulnerable function *decompress_smooth_data()*, triggering the vulnerability.

Secondly, consider *libjpeg-turbo→libtiff→OpenJPEG*. *OpenJPEG* depends on *libtiff* for handling TIFF files. Upon receiving a TIFF file, such as PoC3.tif, it first identifies the file format and preprocesses it using custom code. It then invokes the API function *TIFFReadEncodedStrip()* from *libtiff* to process the data, as shown in Figure 1. During the parsing of embedded JPEG data, *TIFFReadEncodedStrip()* forwards the data to *libjpeg-turbo*, ultimately triggering the vulnerability.

It is important to note that only the original PoC is publicly available, while both downstream PoCs are generated by ChainFuzz. However, generating these PoCs is not easy. We identify three challenges that require careful consideration.

Challenge I: Rising Intra-layer Code and Constraints. As

```
/* Software: libtiff, File: libtiff/tif_jpeg.c */
1  static int JPEGPreDecode(TIFF *tif, ...) {
2      JPEGState *jpeg = JState(tif); //JPEG file data
3      TIFFjpeg_read_header(sp, TRUE);
4      uint32_t seg_width = tif->td_imgwidth;
5      uint32_t seg_height = tif->td_imglength - tif->tif_row;
6      if(jpeg->width != seg_width || jpeg->height != seg_height)
7          return 0;        The JPEG and TIFF data must satisfy specific constraints
8      ...
9  }
```

(a) Rising Intra-layer Codes and Constraints

```
/* Software: OpenJPEG, File: src/bin/jp2/converttif.c */
1  opj_image_t* tiftoimage(TIFF *tif, ...) {
2      uint16_t tiPhoto;
3      TIFFGetField(tif, TIFFTAG_PHOTOMETRIC, &tiPhoto);
4      if(tiPhoto != RGB)
5          return NULL;
6      ...
7  }
/* Software: libtiff, File: libtiff/tif_jpeg.c */
8  int JPEGSetupDecode(TIFF *tif) {
9      JPEGState *jpeg = JState(tif);
10     jpeg->photometric = tif->td_photometric; //==tiPhoto
11     switch(jpeg->photometric) {
12         case YCBCR:
13             jpeg->h_smp = tif->td_ycbcrsubsam[0]
14             jpeg->h_smp = tif->td_ycbcrsubsam[1]
15             break;      Conditional checks in OpenJPEG can
16         default:        affect the checks in libtiff
17             jpeg->h_smp = 1;
18             jpeg->v_smp = 1;
19             break;
20 }}
21 int JPEGPreDecode(TIFF *tif, ...) {
22     JPEGState *jpeg = JState(tif);
23     if(jpeg->h_fac != jpeg->h_smp || jpeg->v_fac != jpeg->v_smp)
24         return 0;
25     ...
26 }
```

(b) Inter-layer Dependency

Figure 2: The sanity checks introduced by each layer.

shown in Figure 1, each dependency layer[1] introduces significant customized code and new branch constraints, resulting in longer and more complex vulnerability exploitation paths. For example, the code snippet in Figure 2 (a) shows an input check introduced by *libtiff*. The condition at line 6 ensures that the *width* and *height* of the JPEG data match the *seg_width* and *seg_height* of the TIFF segment. If the condition is not met, the input is deemed invalid, and the program exits.

Directed grey-box fuzzing (DGF) appears promising for this challenge. DGF [2] aims to test specific target locations without wasting resources on unrelated code and has proven efficient, especially for bug reproduction. Theoretically, one could identify the buggy site in the upstream software as the target and apply DGF to generate a PoC for the downstream software. However, existing DGF approaches [2, 3, 15, 23] primarily focus on exploring paths to vulnerabilities within individual projects. For instance, AFLGo [2] and Beacon [15] calculate distances to the target based on the control-flow graph (CFG). However, constructing a unified CFG for upstream and downstream software, such as *libjpeg-turbo* and *libtiff*, which are distinct programs, presents a challenge.

[1]In this paper, a *layer* refers to a level, i.e., an open-source library, in the hierarchy of software dependencies.

Challenge II: Inter-layer Dependencies. Generating PoCs for *OpenJPEG* involves navigating inter-layer dependencies created by customized checks on TIFF files. For example, *OpenJPEG* verifies whether the *tiPhoto* field is set to RGB (line 4 in Figure 2(b)), which directly influences the execution paths in *libtiff*. In *libtiff*, the *photometric* property of the JPEG data is set to the *td_photometric* of the TIFF data (line 10), which corresponds to *tiPhoto*. It then sets *h_smp* and *v_smp* based on *jpeg→photometric*. At line 23, *libtiff* checks whether *jpeg→h_factor* and *jpeg→v_factor* match *jpeg→h_smp* and *jpeg→v_smp*, respectively. These operations create control and data dependencies between the conditional statements at lines 11 and 23 in *libtiff* and line 4 in *OpenJPEG*.

In the software supply chain, path constraints within individual layers can interact across layers, becoming increasingly complex as the number of layers grows. Overlooking inter-layer dependencies when generating PoCs can lead to conflicts, where satisfying constraints in one layer introduces issues in another. For example, mutating the input field for line 11 may make it unreachable due to the condition check at line 4 in Figure 2(b). However, existing approaches [5,6] typically focus on intra-project constraints and dependencies, making them less effective in supply chain scenarios.

Challenge III: Adapting to Long Supply Chains. Consider a supply chain with $n+1$ layers, where S_i $(0 \leq i \leq n)$ represents each layer of dependency. S_0 is the upstream software containing a known vulnerability, e.g., *libjpeg-turbo*, while S_n is the downstream software requiring verification of the vulnerability's exploitability, e.g., *OpenJPEG*. One potential approach is a top-down strategy that starts from S_n and uses the vulnerability in S_0 as the target for bug reproduction [2]. However, this strategy is computationally expensive and inefficient. If each layer introduces an average of M potential exploitable intra-layer paths, the top-down strategy must consider $O(M^n)$ paths without knowing whether they ultimately reach the vulnerability. This exponential growth in the path space makes it challenging for existing methods, such as DGF, to be applied effectively. Additionally, each layer can introduce customized sanity checks, as shown in Figure 2, further complicating the adaptation of the PoC to satisfy the nested constraints of a long supply chain.

2.2 Our Solution

To tackle the above challenges, we introduce two techniques.

Technique I: Cross-layer Differential Directed Fuzzing. As discussed in Challenge I, constructing a unified CFG for both upstream and downstream software is impractical and imprecise. However, without a CFG, calculating statement distances to targets and performing directed fuzzing becomes challenging. Our solution is to identify specific *waypoints* between the upstream and downstream software, which act as intermediate milestones for all paths that trigger vulnerabilities from the downstream. For example, the function `jpeg_read_scanlines`, as shown in Figure 1, serves as a *waypoint*. We first target these *waypoints* for directed fuzzing within the downstream software. Using these *waypoints*, we segment the execution trace of each input into two parts: the upstream trace (T_u) and the downstream trace (T_d).

In the exploration stage, we prioritize inputs based on two factors: (1) the similarity between T_u and the vulnerable execution trace T_v of the upstream vulnerability PoC, and (2) the diversity of T_d. This input prioritization strategy balances thorough exploration of downstream paths with the assessment of exploitability. In the exploitation stage, we refine inputs derived from the original PoC to generate PoCs for downstream software. By analyzing differences in both *data flow* and *control flow* between T_u and T_v, we apply targeted input mutations to generate effective downstream PoCs.

Technique II: Bottom-up Proof-of-Concept Generation. We propose an efficient bottom-up PoC generation approach consisting of two key steps:

First, we divide the software supply chain, consisting of $n+1$ layers, into n subchains with direct dependencies. Specifically, adjacent layers S_i and S_{i+1} $(0 \leq i < n)$ are considered direct dependencies. Each subchain is handled as an independently explorable subproblem. To thoroughly explore execution paths in each subchain, we identify *waypoints* between S_i and S_{i+1} as targets and apply the strategies from *Technique I* to generate corresponding inputs.

Second, we apply a *bottom-up PoC generation* strategy, progressing step-by-step from S_0 to S_n. PoCs for S_{i+1} are generated based on the PoCs of S_i. At each step, we apply mutation strategies to finely adjust the inputs from S_{i+1} to S_i based on the PoCs for S_i, iteratively repeating this process until we generate PoCs usable for S_n. This divide-and-conquer approach enables ChainFuzz to efficiently tackle the complexities of generating PoCs for long software supply chains.

However, during PoC generation for S_{i+1}, branch constraints in S_{i+1} may conflict with constraints from an earlier layer S_b $(b < i+1)$, causing PoC generation to fail. To tackle this issue, we introduce a *task revisiting* strategy, rolling back to layer S_b to generate a more diverse set of PoCs, which are then used to restart the bottom-up generation from S_b to S_{i+1}.

3 Design and Methodology

In this section, we detail our approach to generating PoCs for downstream software that depends on a vulnerable library.

3.1 Cross-layer Differential Directed Fuzzing

We propose cross-layer differential directed fuzzing, as illustrated in Figure 4. Unlike existing DGF approaches, our method first identifies the *waypoints* (§3.1.1) between the upstream and downstream software and targets them for directed

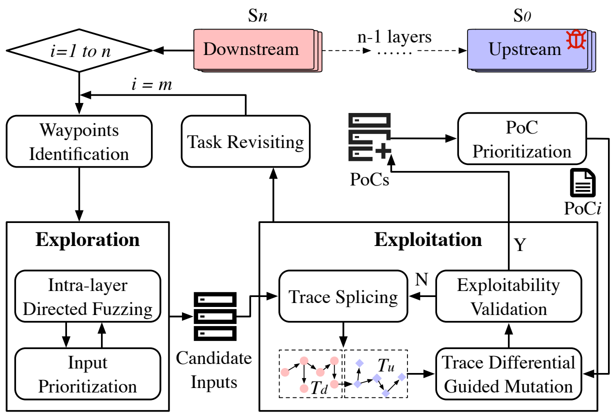


Figure 3: Overall Architecture of CHAINFUZZ.

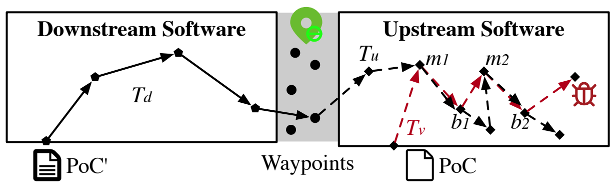


Figure 4: Cross-layer Differential Directed Fuzzing.

fuzzing. These *waypoints* represent critical milestones for all paths that propagate the upstream vulnerability to the downstream software. By targeting the *waypoints*, we only need to construct the control flow graph for the downstream software, significantly reducing complexity. Accordingly, we divide the execution trace of each input into an upstream trace T_u, and a downstream trace T_d. In the exploration stage (§3.1.2), we prioritize inputs to fully explore the *diversity* and *exploitability* of T_d and T_u. In the exploitation stage (§3.1.3), we perform fine-grained mutations of the inputs to satisfy the data and control dependencies between the upstream and downstream software, ultimately triggering the vulnerability.

3.1.1 Waypoint Identification

Directly fuzzing the vulnerable zones of upstream vulnerabilities in software supply chains is often inefficient. We observe that exploitation paths from downstream software to an upstream vulnerability must pass specific program statements, which we define as *waypoints*. By targeting these waypoints, we can perform more efficient and focused directed fuzzing.

Firstly, we define waypoints as critical methods:

- Vulnerable methods like *process_data_simple_main* and *jpeg_read_scanlines* in Figure 1, because the buggy site in *libjpeg-turbo* can only be accessed through them.
- APIs provided by each layer, such as those of *libjpeg-turbo*, because they are the only entry points to its code space. For example, *jpeg_read_scanlines* is one such API, making it a waypoint.

Secondly, we discuss how to identify these waypoints. First, we reproduce the vulnerability using the original PoC to obtain the vulnerable execution trace T_v and the vulnerable function. Then, we construct a call graph of the upstream software using SVF [42] to identify all functions that reach the vulnerable function, forming the set A. Finally, we analyze the downstream software to identify all upstream functions called by it, forming the set B. The intersection of sets A and B yields C, which includes all the waypoints.

However, it's worth noting that set C may be imprecise. First, since we only consider reachability within the call graph and not the control flow, the set C may include functions that cannot actually reach the vulnerable function, resulting in redundancy. Second, due to the use of function pointers, the upstream call graph may lack precision, potentially omitting critical functions and causing incompleteness. We address these flaws in the dynamic testing stage (detailed in §3.1.3).

Running Example. To better understand CHAINFUZZ, we use the motivating example from Section 2.1 as a running example to demonstrate how CHAINFUZZ operates. First, we introduce the identified waypoints. Using SVF, we construct the call graph of *libjpeg-turbo* and attempt to identify waypoints as described. However, due to function pointers, the initial set of identified waypoints is empty. To address this, we consider all functions in *libjpeg-turbo* that are called by *libtiff* as potential waypoints. Through this process, we identify 66 waypoints in *libtiff*. During the exploitation stage (detailed in Section 3.1.3), we exclude 63 of these waypoints and ultimately generate PoCs using only one waypoint: *jpeg_read_scanlines*.

3.1.2 Exploration Stage

In the exploration phase, our goal is to uncover as many potential paths as possible for exploiting upstream vulnerabilities through downstream software. To begin, we target the identified waypoints for directed fuzzing in the downstream software. However, not all paths to waypoints lead to upstream vulnerabilities, resulting in the exploration of redundant paths and reduced efficiency. To address this, we propose a novel input prioritization strategy to focus on more promising paths. Specifically, for inputs that successfully reach the target, we segment their execution traces into two fragments:

- T_u, representing the execution trace fragment within the upstream software.
- T_d, representing the execution trace fragment within the downstream software.

Our approach is based on the idea that to uncover more potential exploitation paths and expose additional attack surfaces, it is essential to thoroughly explore T_d. Simultaneously, to increase the likelihood of triggering the vulnerability in upstream software, T_u should be similar to the execution trace T_v (shown in Figure 4) of the original PoC. To balance these objectives, we prioritize inputs that exhibit a higher similarity to T_v while maintaining rarer T_d, enabling us to explore a diverse range of execution paths in the downstream software and focus on inputs more likely to lead to upstream vulnerabilities. To achieve this, we optimize the distance calculation used in traditional DGF-based approaches, such as AFLGo [2].

We consider three parts to evaluate the score of each seed, as shown in Formula 1: (1) the distance to the targets, (2) the diversity of T_d, and (3) the similarity between T_u and T_v.

$$Score(i) = d(i)^{-1} + r(i) + s(i) \tag{1}$$

The Distance Computation: Typically, the *waypoints* are specific API functions. ChainFuzz first employs static analysis to identify `Call` instructions within the downstream software where the callee corresponds to a waypoint. The basic blocks containing these instructions are designated as targets, denoted as $\psi(b)$. Next, we construct a CFG for the downstream software and compute the distance from each basic block to these targets. During the dynamic testing phase, we conduct directed fuzzing by prioritizing inputs based on their distance to the targets. For each input $i \in Q$, where Q contains all saved inputs, we define the set $\xi(T_d)$, which contains all basic blocks traversed by it. The distance calculation follows the approach proposed in AFLGo [2]. This method ensures that inputs closer to the targets are assigned higher priority.

$$d(i, \psi(b)) = \frac{\sum_{m \in \xi(T_d)} d(m, \psi(b))}{|\xi(T_d)|} \tag{2}$$

The Diversity Computation: In the preprocessing phase, we calculate the distances for all basic blocks that can reach the targets $\psi(b)$ based on the CFG and annotate them accordingly. Let $D = \{m | R(m, \psi(b)) \neq \emptyset\}$ represent the set of all basic blocks with distance annotations, indicating the possibility of reaching the targets. Here, $R(m, \psi(b))$ denotes the set of all targets reachable from basic block m in the CFG. For each $m \in D$, we collect the set $I(m, Q)$, which includes all inputs that reach it. The diversity of m is then determined as:

$$r(m) = \frac{1}{|I(m, Q)|} \tag{3}$$

Therefore, the diversity of input i is determined by:

$$r(i) = \sum_{m \in \xi(T_d) \cap D} r(m) \tag{4}$$

The Similarity Computation: To enhance the efficiency of fuzzing, we perform a coarse-grained *function-level* path similarity evaluation during the exploration stage. Specifically, we begin by reproducing the vulnerability in the upstream software using the original PoC and recording the sequence of functions it executes, which we denote as the target sequence $S(T_v)$. During fuzzing, for each input that reaches the targets, we collect its function execution sequence in the upstream software, referred to as the reference execution sequence $S(T_u)$. Following the approach proposed in VulScope [7], we measure the similarity between two execution sequences by computing their Longest Common Subsequence (LCS). A longer LCS indicates a higher similarity between the sequences, which in turn reflects a greater exploitability of the input. The exploitability of an input is then calculated as:

$$s(i) = \frac{\text{LCS}(S(T_u), S(T_v))}{|S(T_v)|} \tag{5}$$

3.1.3 Exploitation Stage

During the exploitation phase, our objective shifts to refining inputs that can reach upstream libraries and uncovering execution paths capable of triggering upstream vulnerabilities. The core insight behind our approach is that the original PoC of the upstream vulnerability encapsulates the essential vulnerable execution context and bug-triggering conditions, which can be leveraged to generate PoCs for the downstream software. In other words, the downstream and upstream PoCs should exhibit similar execution traces, i.e., T_u and T_v, in the upstream software. Thus, we utilize the similarity of T_u and T_v to guide the seed mutation effectively.

We evaluate the similarity between T_u and T_v at a fine-grained level, considering both control flow and data flow. Figure 4 illustrates the core process of our approach. For each input, we compare its execution trace T_u with T_v, identifying intersection points, e.g., $m1$ and m_2, and bifurcation points, e.g., b_1 and b_2, between the two traces. During each round of mutation, we select a bifurcation point and apply one of our mutators. Our approach consists of two main components:

Byte-level Dynamic Taint Analysis. We use taint analysis to record execution traces and map input fields to their corresponding constraints for the convenience of input mutation.

First, for each input, we track its basic block execution trace within the upstream software, recording the sequence of executed basic blocks. Additionally, we employ byte-level dynamic taint analysis, treating the PoC as the taint source and tracing its propagation within the upstream program. Each tainted variable is assigned a $\langle \text{start}, \text{end} \rangle$ label, indicating the offset of the corresponding field within the input. For every instruction with tainted operands, we log the associated tainted information. Since a program statement may execute multiple times, introducing new tainted data, the tainted information for a variable is stored as a list of labels.

Second, we conduct function-level taint aggregation to identify the input fields processed by each function. Using byte-level dynamic taint analysis, we determine the tainted information associated with each statement. We then aggregate this tainted information upward to the function level, where each function is assigned the composite input processed by its constituent statements.

Figure 5 illustrates an example of our taint aggregation strategy. In the code snippet, the file pointer p in function *foo()* serves as the taint source. The function *foo()* invokes *read2bytes()* and *read4bytes()* to read two and four bytes from

```
0  void foo(FILE *p) {
1      int width = read2bytes(p);
2      int height = read4bytes(p);
3      if (width * height > MagicNum) {    return;    }}
4  int read2bytes(FILE *p) {
5      unsigned char temp[2]
6      fread(temp, sizeof(unsigned char), 2, p);
7      return temp[0] << 8 | temp[1];
8  }
9  int read4bytes(FILE *p) {
10     unsigned char temp[4]
11     fread(temp, sizeof(unsigned char), 4, p);
12     return temp[0] << 24 | temp[1] << 16 | temp[2] << 8 | temp[3];
13 }
```

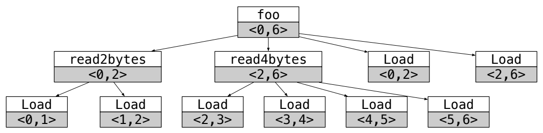


Figure 5: Function level taint aggregation.

the file, respectively. The tree structure demonstrates the taint aggregation process. The leaf nodes represent the program statements whose operands have taint labels. For instance, at line 7, the code snippet loads variables *temp[0]* and *temp[1]*, which have taint labels $\langle 0,1\rangle$ and $\langle 1,2\rangle$, respectively. Consequently, we aggregate the taint labels of these two `Load` statements to the function *read2bytes()* as the input region processed within it. Similarly, the input region processed in the function *foo()* is identified as $\langle 0,6\rangle$.

Trace Differential Guided Mutation. Using collected execution traces and taint analysis results, we propose a mutation strategy guided by trace differentials. This approach efficiently refines inputs generated during the exploration phase, enabling them to trigger upstream vulnerabilities. Below, we introduce the customized mutators of CHAINFUZZ.

Mutator# I: Coarse-grained Input Splicing. In §2.1, we observed that downstream software often relies on upstream libraries to process specific portions of data. For instance, *libtiff* depends on *libjpeg* to handle JPEG data embedded within TIFF files. Given this characteristic, one intuitive approach is to divide each input into two parts: the portion processed by the downstream software and the portion processed by the upstream software. By mapping the upstream-processed portion to the content in the original PoC, we can efficiently mutate the input. Expanding on this *input mapping* idea, we propose two mutation strategies for input splicing: coarse-grained input splicing and fine-grained field replacement.

First, we introduce coarse-grained input splicing. Using the taint aggregation strategy, we identify the data regions processed within each function along the execution trace for each input. For input i, we select a function f executed in the upstream software, where the taint label for f is $\langle \text{start}_i, \text{end}_i\rangle$. If the corresponding taint label for f when processing the original PoC is $\langle \text{start}_p, \text{end}_p\rangle$, we replace the region $\langle \text{start}_i, \text{end}_i\rangle$ in input i with the region $\langle \text{start}_p, \text{end}_p\rangle$ from the PoC.

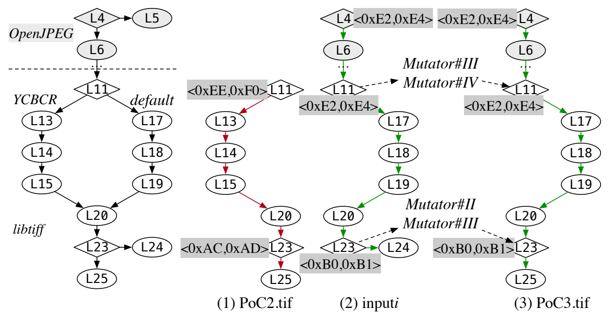
Mutator# II: Fine-grained Field Replacement. We only consider input fields processed in conditional statements. Specifically, our *fine-grained field replacement* strategy operates as follows: For each input i, we collect all the conditional statements it executes in the downstream and upstream software, along with their taint labels. For each upstream conditional statement s with taint label $\langle \text{start}_i, \text{end}_i\rangle$, we collect its taint label $\langle \text{start}_p, \text{end}_p\rangle$ when processing the original PoC. If $\langle \text{start}_i, \text{end}_i\rangle$ is not processed by any downstream conditional statements, we replace the region $\langle \text{start}_i, \text{end}_i\rangle$ in input i with the region $\langle \text{start}_p, \text{end}_p\rangle$ in the PoC. It is worth noting that start_i and start_p, as well as end_i and end_p, are usually different since they correspond to distinct inputs.

Mutator# III: Intra-layer Mutation. By employing *Mutator# I* and *Mutator# II*, we can significantly increase the likelihood of inputs reaching the buggy site. However, successfully triggering the vulnerability often requires additional fine-grained adjustments to align their execution traces with the original PoC.

Specifically, we compare the execution traces T_u and T_v to locate the first matching basic block, e.g., m_1 in Figure 4. Starting from m_1, we iteratively compare T_u and T_v to identify the first unmatched basic block, e.g., b_1. Then, we analyze the intra-procedure control flow graph to locate the immediate predecessor of the unmatched basic block. Within this immediate predecessor, we extract the condition variables associated with the branch decision, such as variables used in `CMP` or `SWITCH`. These condition variables, controlling the branch outcome, are crucial for subsequent adjustments to ensure synchronization between the execution traces. Based on our taint analysis, we can locate the critical input bytes in the input that affect the runtime values of the condition variables. We then focus on mutating these critical bytes to increase the similarity between T_u and T_v. In this process, we continuously search for the next unmatched basic block and apply the above procedure, gradually aligning T_u with T_v.

Mutator# IV: Cross-layer Mutation. As discussed in §2.1, control and data dependencies exist between conditional statements of different layers. When mutating identified critical bytes, the upstream conditional statement may become unreachable because the downstream branch is not satisfied. To solve this, we propose a cross-layer mutation strategy.

For a selected input i, we collect its execution trace T_d and T_u. Suppose we need to mutate the input field $\langle \text{start}_i, \text{end}_i\rangle$ to satisfy the constraint of an upstream conditional statement. After each mutation, we generate a new input and collect the corresponding trace T'_d and T'_u, aiming for T'_d to be the same as T_d and T'_u to be more similar to T_v than T_u. However, if the mutated field is also checked in a downstream branch, the new field value may not satisfy the branch constraint, resulting in T'_d being different from T_d and the input being unable to enter the upstream software. To address this, we compare traces T'_d and T_d to locate the first different basic block and its immediate predecessor, following the approach outlined in *Mutator# III*. From there, we extract the input field $\langle \text{start}'_i, \text{end}'_i\rangle$ for the condition variables in the predecessor. If $\langle \text{start}'_i, \text{end}'_i\rangle$ is the same as $\langle \text{start}_i, \text{end}_i\rangle$, we continue attempting to mutate



(a) The CFG of Figure 2(b)   (b) Mutate input*i* according to the execution trace of PoC2.tif

Figure 6: Trace Differential Guided Mutation Strategy.

the field, as its current value fails to satisfy both upstream and downstream branch constraints. However, if $\langle \text{start}_i', \text{end}_i' \rangle$ is different from $\langle \text{start}_i, \text{end}_i \rangle$, we focus on mutating the input bytes within the range of their difference.

Waypoint Optimization. As discussed in §3.1.1, the identified *waypoints* have issues with redundancy and incompleteness. To address these issues, we propose two mitigation strategies. First, to eliminate redundant waypoints, we identify inputs whose execution trace T_u cannot be fitted to the vulnerable trace T_v during the exploitation stage. We then analyze which waypoints these inputs reached and identify the most frequent ones as redundant. Second, to discover missed waypoints, we identify inputs that enter the upstream software but do not pass through any known waypoints during the exploration stage. We then add the entry point functions of these inputs to our waypoint set. During on-the-fly identification, we defer verifying if new waypoints can reach vulnerable functions, as this would require a precise call graph. Instead, we evaluate their effectiveness and remove redundant waypoints during dynamic fuzzing, as described above.

Running Example. Figure 6 shows how CHAINFUZZ's mutators generate the PoC for *OpenJPEG* (PoC3.tif) based on the execution trace of the PoC for *libtiff* (PoC2.tif). Figure 6(a) shows a simplified cross-layer control flow graph of the code snippet from Figure 2(b). Figure 6(b) displays the execution traces of three inputs: (1) the trace of PoC2.tif; (2) the trace of input$_i$ generated while testing *OpenJPEG* with CHAINFUZZ; (3) the trace of PoC3.tif, generated by mutating input$_i$ based on the differences in its trace compared to PoC2.tif.

CHAINFUZZ starts by identifying intersections and bifurcation points between the traces of PoC2.tif and input$_i$. The first intersection, L11, corresponds to the `SWITCH` instruction at line 11 of Figure 2(b). After L11, PoC2.tif and input$_i$ diverge into different cases. To align the traces, CHAINFUZZ uses taint analysis to determine the taint label of the variable at line 11. For PoC2.tif, the taint label is $\langle 0\text{xEE}, 0\text{xF0} \rangle$, while for input$_i$, it is $\langle 0\text{xE2}, 0\text{xE4} \rangle$. Using ***Mutator# III***, CHAINFUZZ randomly mutates the input range $\langle 0\text{xE2}, 0\text{xE4} \rangle$ in input$_i$ to match the case executed by PoC2.tif. However, because this region is also checked at line 4 in *OpenJPEG*, altering its value causes the generated inputs to execute line 5, preventing them from entering *libtiff*. Next, CHAINFUZZ applies ***Mutator# IV***, enumerating possible values for $\langle 0\text{xE2}, 0\text{xE4} \rangle$ in input$_i$. Only one value successfully allows the inputs to enter *libtiff*. Despite this, the traces diverge after line 11, so CHAINFUZZ searches for the next intersection at L20. After L20, the traces diverge again after L23. At this point, CHAINFUZZ uses ***Mutator# II***, replacing the range $\langle 0\text{xB0}, 0\text{xB1} \rangle$ in input$_i$ with the corresponding range $\langle 0\text{xAC}, 0\text{xAD} \rangle$ from PoC2.tif. However, the generated input still fails to execute L25. Finally, CHAINFUZZ applies ***Mutator# III***, randomly mutating the range $\langle 0\text{xB0}, 0\text{xB1} \rangle$ in input$_i$. This process ultimately generates PoC3.tif.

3.2 Bottom-up Proof-of-Concept Generation

CHAINFUZZ can generate PoCs for direct vulnerable dependencies using *cross-layer differential directed fuzzing*. In this section, we introduce its bottom-up PoC generation approach.

3.2.1 Bottom-up PoC Generation

In bottom-up PoC generation, the objective is to generate the final PoC$_n$ for S_n, where PoC$_{i+1}$ for layer S_{i+1} is generated based on PoC$_i$ for layer S_i. We have two optional strategies for this process. The first strategy generates PoC$_{i+1}$ immediately after obtaining a usable PoC$_i$ for layer S_i, while the second waits until a sufficient number of PoC$_i$s have been generated for layer S_i before proceeding to S_{i+1}. Each strategy has its own advantages and drawbacks. The first strategy prioritizes efficiency, enabling faster generation of the final PoC$_n$. However, it may encounter limitations if the PoC$_i$ cannot satisfy specific branch constraints in certain layers. In contrast, the second strategy ensures a diverse set of usable PoC$_i$s, improving the likelihood of generating a usable PoC$_n$. However, this approach requires more time for a comprehensive exploration of each layer, leading to lower efficiency.

To combine the strengths of both strategies, we propose a goal-oriented bottom-up approach. First, we introduce a PoC prioritization strategy. After generating PoCs for layer S_i, we evaluate their scores and prioritize those with higher scores to guide the generation of PoCs for S_{i+1}. Second, we propose a task revisiting strategy. If we consistently encounter situations where a particular branch constraint in layer S_b $(0 \leq b < i+1)$ cannot be satisfied during the generation of PoC$_{i+1}$, we revisit the task. Specifically, we roll back to layer S_b, regenerate additional PoC$_b$s, and attempt to generate subsequent PoCs based on the newly generated ones.

PoC prioritization. After generating PoCs for layer S_i, we prioritize them to select the most promising PoC$_i$ to guide the generation of PoCs for layer S_{i+1}. We evaluate PoC$_i$ from two perspectives. In the downstream direction, we assess the degree of similarity between PoC$_i$ and the inputs generated

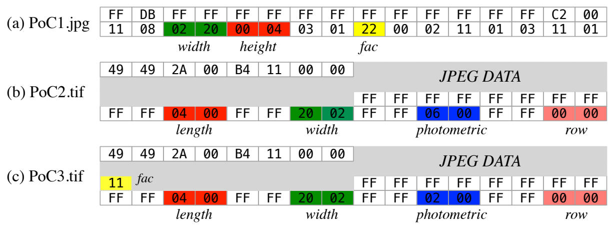


Figure 7: The PoCs for our motivating example in Figure 1.

from S_{i+1} to S_i, i.e., inputs in Q_{i+1}. The more inputs with execution paths similar to PoC$_i$, the higher its score. In the upstream direction, suppose PoC$_i$ is generated under the guidance of PoC$_{i-1}$ of layer S_{i-1}. We evaluate the product of two factors: the similarity between the execution paths of PoC$_i$ and PoC$_{i-1}$ and the score of PoC$_{i-1}$. A higher product indicates a higher score for PoC$_i$. Specifically, the score of PoC$_i$ can be calculated using the following formula:

$$Score(PoC_i) = \frac{\sum_{i \in Q(i+1)} s(i)}{|Q(i+1)|} + Score(PoC_{i-1}) \cdot s(PoC_i) \quad (6)$$

Task revisiting. We perform *task revisiting* when attempts to generate PoCs for the layer S_{i+1} consistently fail. Firstly, we identify the bottleneck for the current layer S_{i+1} where generating the corresponding PoC$_{i+1}$ encounters significant difficulty. As discussed, inter-layer dependencies can cause branch constraints in layer S_{i+1} to conflict with constraints in intermediate dependent layers. Such conflicts make it challenging to generate usable PoCs for S_{i+1}. To address this issue, we collect the temporary inputs generated during attempts to generate PoCs for S_{i+1} and analyze the lowest layer of dependent software reachable by these inputs. This layer is identified as the bottleneck layer S_b. Simultaneously, we determine the branch constraint conditions in S_b that these inputs consistently fail to satisfy. Secondly, we roll back to the layer S_b and attempt to generate new PoC$_b$s for S_b that trigger upstream vulnerabilities without passing through the identified conflicting branch constraints. Thirdly, after successfully generating new PoC$_b$s, we proceed to generate new PoCs for layers S_{b+1} through S_{i+1} based them.

Running Example. Figure 7 highlights partial critical data in the PoC files for our motivating example. Compared to PoC1.jpg, which is in JPEG format, PoC2.tif is a TIFF file that embeds the JPEG data and must satisfy specific constraints. For example, the *width* field in TIFF data must match the *width* field in the embedded JPEG data, as checked at line 6 in Figure 2(a). In contrast, PoC3.tif introduces additional constraints. The *photometric* field in PoC3.tif must be set to 0x02 to satisfy the branch constraint in *OpenJPEG* (line 4, Figure 2(b)), and the *fac* field in the embedded JPEG data must be 0x11 to satisfy the branch constraint in *libtiff* (line 23, Figure 2(b)). These highlighted data fields are crucial to ensuring that all three levels of PoCs successfully trigger the same upstream vulnerability.

4 Evaluation

In this section, we evaluate the effectiveness of CHAINFUZZ in generating PoCs to exploit upstream vulnerabilities.

Prototype Implementation. We implemented CHAINFUZZ for C/C++ software with 2.3k lines of C/C++ code, 1.2k lines of Python code, and 1.6k lines of Rust code. Dynamic taint analysis was implemented based on the LLVM Dataflow Sanitizer [24]. To collect runtime taint information and the corresponding field offset of inputs, we instrumented specific instructions, e.g., `Load`, `Call`, `Cmp`, using an LLVM Pass. We implemented the fuzzing component based on AFLGo [2].

Dataset Collection. As discussed earlier, CHAINFUZZ is a directed, fuzzing-based approach that takes an original PoC and a software supply chain as input to generate PoCs applicable to downstream software. While existing benchmarks [10, 14, 26] evaluate the effectiveness of direct fuzzing approaches, they typically include only vulnerable upstream software. However, these benchmarks often lack PoCs for the vulnerabilities and omit information about downstream software that depends on the upstream. To address this limitation, we constructed a custom dataset to evaluate CHAINFUZZ. Firstly, we selected nine popular upstream software, as shown in Figure 8, based on the following criteria:

1. The software has been widely tested by existing fuzzing approaches [2, 8].
2. Published vulnerability reports were available.
3. Downstream software depending on the upstream software could be identified. To determine these dependencies, we used two methods. (1) Reviewing the documentation of the upstream software to confirm its usage in the downstream software. (2) Examining the configuration files of the downstream software to extract references to the upstream software.

Secondly, for each upstream software, we collected vulnerability reports from the National Vulnerability Database (NVD) [31] and public security issues from their Git repositories. We filtered these reports based on the following criteria:

1. *Availability of Proof-of-Concept input*. We selected vulnerabilities with publicly available PoC inputs.
2. *Reproducibility*. Although some vulnerabilities have public PoCs, reproduction can be challenging due to incomplete vulnerability reports [28]. Common issues include unspecified vulnerable versions, missing configuration details, or undefined operating system requirements. We attempted to reproduce all vulnerabilities and excluded those we could not reproduce from our dataset.

Thirdly, for each vulnerable upstream software, we selected the downstream software that depends on it, as shown in Fig-

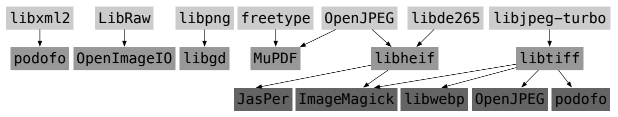


Figure 8: The supply chains in our dataset.

ure 8. To comprehensively evaluate the capability of ChainFuzz, we mainly tested it with two different software supply chain lengths. (1) Length of 2: For example, *ImageMagick* directly depends on *libheif*. In this case, we used ChainFuzz to generate PoCs for *ImageMagick* that can trigger the vulnerabilities in *libheif*. (2) Length of 3: For example, *libtiff* depends on *libjpeg-turbo*, and *OpenJPEG* depends on *libtiff*. In this case, we used ChainFuzz to generate PoCs for *libtiff* and *OpenJPEG* that could trigger the vulnerabilities in *libjpeg-turbo*. This setup allowed us to assess ChainFuzz's ability to validate the exploitability of vulnerabilities in transitive dependencies across long software supply chains. For each downstream software, we used its latest version.

In summary, our dataset contains 189 real-world vulnerabilities, 20 supply chains (shown in Figure 8), and 554 unique pairs of $\langle \text{vulnerability}, \text{supply-chain} \rangle$, covering 16 widely used software. These software programs can process more than 12 different file formats.

Baselines. To the best of our knowledge, no existing approaches focus on generating PoCs for C/C++ downstream software. The work most similar to ChainFuzz is direct greybox fuzzing. For our evaluation, we selected AFLGo [2], a state-of-the-art and widely recognized directed fuzzing approach. While researchers have proposed many DGF approaches, such as Beacon [15] and Titan [16], we find these methods unsuitable for generating PoCs for upstream vulnerabilities. This is because their instrumentation and static analysis modules are designed for runnable binary programs, e.g., djpeg, rather than entire upstream software libraries, e.g., *libjpeg*, which are typically depended on by the downstream. We excluded FuzzGuard [50], Hawkeye [4], MC2 [40], CAFL [22], DSFuzz [23], and PDGF [49] from our baselines because these tools were not publicly available at the time of writing. AFLGo calculates and instruments the distance of each basic block to specified targets based on the program's CFG. To thoroughly assess AFLGo's ability to trigger upstream vulnerabilities, we used two different configurations: AFLGo-Up and AFLGo-Down. For AFLGo-Up, we manually reproduced the upstream vulnerability and identified the buggy site as the target. For AFLGo-Down, we identified waypoints between the downstream and upstream based on ChainFuzz and used these as targets. In addition to AFLGo, we included AFL++ [12], a general fuzzing approach that integrates various fuzzing research and ranks highly among fuzzing tools according to Fuzzbench [26]. Since most software in our dataset processes structured inputs, we also included NestFuzz [8], an input structure-aware fuzzing approach, as one of our baselines.

Test Bed and Configuration. All experiments were conducted on a Ubuntu 20.04 server with AMD EPYC 7513 CUPs (128 cores) and 1024 GB RAM. Each testing task was executed on an individual virtual machine with 16GB of memory and 2 CPU cores. We enabled Address Sanitizer [39] for each fuzzing campaign to detect vulnerabilities. When a vulnerability was discovered, we manually confirmed whether the original upstream vulnerability had been triggered. To ensure fairness across all fuzzing tools, we used the same seeds for fuzzing the downstream software, including the original PoC and one valid input specific to the downstream software.

4.1 Effectiveness of ChainFuzz

4.1.1 Exploitability of the vulnerabilities in our dataset

We first investigated whether the upstream vulnerabilities in our dataset could be exploited downstream. Three experts, each with six years of experience in software security, participated in the analysis. Two experts independently evaluated the exploitability of the vulnerabilities by manually crafting the original PoCs for the corresponding supply chains. A third expert reviewed their findings, achieving a Cohen's Kappa [45] coefficient of 0.962, indicating a strong level of agreement. Finally, we categorized the $\langle \text{vulnerability}, \text{supply-chain} \rangle$ pairs in our dataset into three categories:

- Case#1 (3.25%): the PoC triggered the same upstream vulnerability via the downstream software.
- Case#2 (11.91%): the PoC, after mutation, triggered the same upstream vulnerability via the downstream.
- Case#3 (84.83%): the upstream vulnerability could not be triggered.

Based on our results, Case#3 accounted for the highest percentage. There are two primary reasons: (1) the vulnerable function in the upstream software is unreachable in the downstream, and (2) the conditions necessary to trigger the upstream vulnerability cannot be satisfied in the downstream, often due to customized configurations like environment or global variables in the upstream software. Table 5 summarizes the details of the exploitable vulnerabilities in our dataset.

4.1.2 Evaluation of ChainFuzz's effectiveness

Experiment I: Generating PoCs for Exploitable Vulnerabilities. In this experiment, we evaluated the effectiveness of ChainFuzz in generating PoCs for downstream software to trigger exploitable upstream vulnerabilities. Using the results from § 4.1.1, we established a ground truth of 66 unique $\langle \text{vulnerability}, \text{supply-chain} \rangle$ pairs categorized as Case#2.

For each vulnerability, we employed ChainFuzz and our baselines to generate PoCs for downstream software that directly or transitively depend on the corresponding upstream

Table 1: The effectiveness of ChainFuzz, AFLGo-Up, AFLGo-Down, NestFuzz, and AFL++ in generating PoCs for upstream vulnerabilities in the downstream.

CVE ID	**Downstream#1**	**μTTE(h)**					**Downstream#2**	**μTTE(h)**				
		ChainFuzz	AFLGo-Up	AFLGo-Down	NestFuzz	AFL++		ChainFuzz	AFLGo-Up	AFLGo-Down	NestFuzz	AFL++
CVE-2016-10506	libheif	2m/45s	✗	✗	✗	✗	JasPer	2m/40s	✗	✗	✗	✗
							ImageMagick	2m/34s	✗	✗	✗	✗
CVE-2019-6988	libheif	4m/13s	✗	✗	✗	✗	JasPer	3m/1s	✗	✗	✗	✗
							ImageMagick	3m/5s	✗	✗	✗	✗
CVE-2023-39328	libheif	3m/05s	✗	✗	✗	✗	JasPer	5m/32s	✗	✗	✗	✗
							ImageMagick	5m/11s	✗	✗	✗	✗
Issue#389	libheif	3m/12s	✗	✗	✗	✗	JasPer	4m/43s	✗	✗	✗	✗
							ImageMagick	4m/50s	✗	✗	✗	✗
Issue#393	libheif	3m/07s	✗	✗	✗	✗	JasPer	5m/25s	✗	✗	✗	✗
							ImageMagick	5m/40s	✗	✗	✗	✗
Issue#394	libheif	3m/01s	✗	✗	✗	✗	JasPer	5m/09s	✗	✗	✗	✗
							ImageMagick	5m/04s	✗	✗	✗	✗
Issue#399	libheif	3m/18s	✗	✗	✗	✗	JasPer	4m/52s	✗	✗	✗	✗
							ImageMagick	5m/02s	✗	✗	✗	✗
Issue#1501	libheif	4m/11s	✗	✗	✗	✗	JasPer	4m/23s	✗	✗	✗	✗
							ImageMagick	4m/31s	✗	✗	✗	✗
Issue#1505	libheif	4m/20s	✗	✗	✗	✗	JasPer	5m/07s	✗	✗	✗	✗
							ImageMagick	5m/07s	✗	✗	✗	✗
CVE-2020-21600	libheif	1h/7m/47s	✗	✗	✗	✗	JasPer	1h/16m/48s	✗	✗	✗	✗
							ImageMagick	58m/11s	✗	✗	✗	✗
CVE-2021-29390	libtiff	24m/36s	✗	✗	8h/16m/49s (1/5)	7h/45m/36s (2/5)	Podofo	31m/27s	✗	✗	✗	✗
							libwebp	22m/14s	✗	✗	✗	13h/10m/48s (3/5)
							OpenJPEG	2h/19m/16s	✗	✗	✗	✗
							ImageMagick	22m/46s	✗	✗	✗	16h/14m/45s (2/5)
CVE-2022-43239	libheif	2m/24s	✗	✗	✗	✗	JasPer	7m/17s	✗	✗	✗	✗
							ImageMagick	10m/13s	✗	✗	✗	✗
CVE-2022-43252	libheif	2m/35s	✗	✗	✗	✗	JasPer	4m/15s	✗	✗	✗	✗
							ImageMagick	6m/5s	✗	✗	✗	✗
CVE-2023-24752	libheif	6m/36s	✗	✗	✗	✗	JasPer	6m/13s	✗	✗	✗	✗
							ImageMagick	6m/15s	✗	✗	✗	✗
CVE-2023-24751	libheif	4m/49s	✗	✗	✗	✗	JasPer	10m/15s	✗	✗	✗	✗
							ImageMagick	7m/45s	✗	✗	✗	✗
CVE-2023-49468	libheif	1m/47s	55m/11s	55m/09s	✗	2m/24s	JasPer	2m/25s	55m/16s	55m/13s	✗	2m/24s
							ImageMagick	4m/16s	1h/52m/43s	1h/52m/45s	✗	1m/45s
CVE-2023-25221	libheif	4m/33s	✗	✗	✗	✗	JasPer	7m/49s	✗	✗	✗	✗
							ImageMagick	7m/30s	✗	✗	✗	✗
CVE-2024-31619	ImageMagick	1m/11s	4m/13s	2m/27s	1m/14s	0m/35s						
CVE-2016-10506	MuPDF	1m/45s	✗	✗	✗	2h/3m/2s						
CVE-2019-6988	MuPDF	1m/45s	✗	✗	✗	2h/3m/2s						
CVE-2023-39328	MuPDF	1m/23s	✗	✗	✗	✗						
Issue#389	MuPDF	0m/42s	✗	13h/30m/47s (4/5)	6h/25m/56s (1/5)	5h/8m/9s						
Issue#393	MuPDF	0m/37s	5h/21m/12s (2/5)	9h/56m/33s (2/5)	✗	42m/42s						
Issue#394	MuPDF	0m/35s	21m/1s	22m/46s	8h/12m/56s	6m/9s						
Issue#399	MuPDF	1m/54s	✗	✗	✗	4h/5m/17s						
Issue#1501	MuPDF	1m/18s	✗	✗	✗	✗						
Issue#1505	MuPDF	1m/20s	✗	✗	✗	✗						
CVE-2022-0561	Podofo	2m/52s	✗	✗	✗	✗						
CVE-2022-0562	Podofo	4m/29s	✗	✗	✗	✗						
CVE-2022-0908	Podofo	3m/13s	✗	✗	✗	✗						

[1] ✗ means that the PoC is not generated within the 24-hour limit.

software. Each experiment was repeated five times with a 24-hour timeout. We recorded the first time each tool generated a downstream PoC that successfully triggered the upstream vulnerability (TTE) and calculated the average of these times (μTTE). If no PoCs were generated within 24 hours across all five attempts, the time is marked as ✗. If a tool did not consistently generate PoCs in all five attempts, this is also indicated. As shown in Table 1, ChainFuzz was the only approach capable of generating PoCs for all vulnerabilities, accomplishing this task in significantly less time. In contrast, the baseline tools were effective only for certain vulnerabilities, and they were less reliable and required considerably more time. Interestingly, ChainFuzz occasionally generated PoCs for Downstream#2 faster than for Downstream#1. For example, it took 1h/7m/47s for *libheif* while only 58m/11s for *ImageMagick* for the vulnerability CVE-2020-21600. This discrepancy occurs because PoCs generated for Downstream#1 could sometimes be directly used in Downstream#2 to trigger the upstream vulnerability. In some cases, baselines outperformed ChainFuzz. For example, AFL++ generated downstream PoCs faster than ChainFuzz for CVE-2024-31619 in *libheif*. Manual analysis revealed that while the original PoC was unsuitable for *ImageMagick*, a critical bit mutation made it usable. AFL++ could rapidly generate downstream PoCs using mutation strategies such as bit flipping.

Experiment II: Analyzing Case#3 Pairs. A key strength of ChainFuzz is its ability to detect subtle vulnerabilities that may escape manual code review, facilitating the validation of hard-to-detect upstream vulnerabilities. In this experiment, we focused on the remaining 84.83% of $\langle$vulnerability, supply-chain$\rangle$ pairs labeled unexploitable by experts to evaluate labeling accuracy and uncover potential overlooked vulnerabilities. Specifically, we randomly selected 45 pairs from this category, ensuring they covered all upstream libraries listed in Figure 8. Using ChainFuzz, we attempted to generate PoCs for the corresponding downstream software and classified the results into the following three categories:

- Unreachable (22/45): upstream vulnerable function inaccessible downstream, e.g., CVE-2023-2804.
- Reachable (22/45): function accessible but vulnerability cannot trigger, e.g., CVE-2023-2731.
- Exploitable (1/45): vulnerability verified exploitable with the help of ChainFuzz, e.g., CVE-2023-1729.

Table 2: The performance of ChainFuzz compared to SCA and reachability analysis approaches.

Tools	ChainFuzz	CCScanner	Reachability
TP	66	40	7
FN	1	27	60
TN	44	14	38
FP	0	30	6

The detailed results are summarized in Table 6. Notably, ChainFuzz helped identify CVE-2023-1729 as exploitable downstream. This is a vulnerability in *libraw*, a dependency of *OpenImageIO*. While ChainFuzz generated inputs that reached the vulnerable code, they failed to satisfy the exploit conditions. Further manual analysis of the fuzzing results revealed that exploiting this vulnerability required enabling a specific option in *OpenImageIO*'s configuration file during fuzzing. This highlights the limitations of manual analysis and the value of ChainFuzz in vulnerability assessment.

Experiment III: False Positive and False Negative. In this experiment, we evaluated the false positives and negatives of ChainFuzz compared to SCA and reachability analysis approaches. We established the ground truth using the vulnerabilities listed in Table 1 and Table 6. The ground truth consists of 67 exploitable $\langle$vulnerability, supply-chain$\rangle$ pairs and 44 unexploitable pairs. For comparison, we selected CCScanner [44], a dependency detection tool for C/C++ libraries. For each pair in the ground truth, we used CCScanner to detect the dependency of the downstream library. If CCScanner established the supply chain, we considered it as marking the corresponding vulnerability as exploitable. Additionally, we implemented a reachability analysis approach following existing approaches [47]. Specifically, we first constructed the function call graphs for the vulnerable version of the upstream software and the downstream software. Second, we merged the call graphs based on shared function nodes, i.e., the upstream functions called by the downstream software. Third, we searched the merged call graph to determine whether a function-level reachable path exists from the downstream program to the upstream vulnerable function. If a reachable path was found, we considered it as marking the corresponding vulnerability as exploitable. Table 2 summarizes the results.

CCScanner generated high false positives and negatives because it flagged all upstream vulnerabilities as exploitable while failing to identify dependencies for certain downstream software, such as *MuPDF*. Similarly, the reachability analysis approach exhibited high false positives and negatives, as it could not verify whether upstream exploit conditions were satisfied and struggled to construct precise call graphs due to the presence of function pointers. In contrast, ChainFuzz generates concrete PoCs for exploitable upstream vulnerabilities, achieving significantly lower false positives and negatives.

Experiment IV: Adapting Long Supply Chains. In this experiment, we evaluated the ability of ChainFuzz to generate PoCs for long supply chains. First, we studied the average size of supply chains in real-world contexts. We analyzed 6,584 supply chains across 1,386 libraries in Ubuntu-20.04, collected using *apt-rdepends*. The results indicated that most downstream libraries (66.84%) had chain lengths under four levels, with three-level chains being the most common (38.59%). Additionally, 87.03% of libraries had chain lengths under five levels. Based on the results, we selected four-level supply chains as representative examples of long supply chains to evaluate ChainFuzz's effectiveness. We then identified the supply chain: *libde265*→*libheif*→*JasPer*→*FreeImage*. Using the vulnerabilities listed in Table 5, we constructed seven unique $\langle$vulnerability, supply-chain$\rangle$ pairs. ChainFuzz successfully generated PoCs for all supply chains in under 30 minutes. Notably, the downstream PoCs for *libheif*, *JasPer*, and *FreeImage* were distinct for each vulnerability, demonstrating the effectiveness of ChainFuzz in handling long supply chains.

Table 3: Zero-day vulnerabilities found by ChainFuzz when testing the n-day upstream vulnerabilities.

Known CVE	Upstream	Downstream	Zero-day	Status
CVE-2019-6988	OpenJPEG	JasPer	Out of Memory	Reported
		ImageMagick	Heap overflow	Reported
		libheif	Segmentation Fault	Reported
		libheif	Out of Memory	Reported
CVE-2022-43252	libde265	JasPer	NULL Pointer	Confirmed
CVE-2023-0802	libtiff	ImageMagick	Heap Overflow	Confirmed
		ImageMagick	Memory Leak	Confirmed
		ImageMagick	Heap Overflow	Assigned

4.1.3 Find Zero-day Vulnerabilities

While ChainFuzz's primary goal is to generate PoCs for downstream software that trigger upstream vulnerabilities, it unexpectedly uncovered eight zero-day vulnerabilities during our experiments (Table 3). Unlike vulnerabilities found by existing fuzzing tools, these zero-day vulnerabilities involve interactions across multiple software components. For instance, CVE-2022-43252, a vulnerability in *libde265*, affects *libheif*, which subsequently affects *JasPer*. While using ChainFuzz to trigger this vulnerability in *JasPer*, we found a NULL pointer dereference in *JasPer* due to insufficient return value checks when calling *libheif*'s API. We reported this issue to the *JasPer* developers and assisted in its resolution.

These zero-day vulnerabilities in Table 3 can be categorized into two types based on their buggy sites. First, the buggy sites are located in similar code logic within the upstream software as published n-day vulnerabilities. This occurs primarily because the downstream software invokes different upstream functions than those in the original vulnerable execution trace. Second, the buggy sites are in the downstream software, often due to improper use of upstream API functions. ChainFuzz effectively identifies these two types of vulnerabilities through its trace differential guided mutation

Table 4: The number of inputs that can reach the waypoints or vulnerable function (VF) generated by different tools.

Tools	ChainFuzz	AFLGo-Up	AFLGo-Down	NestFuzz	AFL++
Waypoints	11416(49.81%)	799(67.77%)	811(72.21%)	2183(11.93%)	2090(22.49%)
VF	9776(42.65%)	8(0.84%)	8(0.71%)	262(1.43%)	219(2.35%)

and waypoint-guided cross-layer fuzzing techniques.

4.2 Component-wise Analysis

In this experiment, we assessed the effectiveness of our design choices. ChainFuzz benefits from waypoint-directed fuzzing to thoroughly explore intra-layer paths and trace differential guided mutation to generate downstream PoCs. To evaluate the contribution of each component, we developed two variants. ChainFuzz $_{exp}$ disables waypoint identification and directly targets the buggy site of the upstream vulnerability to evaluate the exploration stage. ChainFuzz $_{expl}$ disables the customized mutators from Section 3.1.3 and uses only the mutation strategies implemented by AFLGo to evaluate the exploitation stage. We then compared the effectiveness of ChainFuzz $_{exp}$ and ChainFuzz $_{expl}$ in generating PoCs for the upstream vulnerabilities listed in Table 5. Compared with ChainFuzz, ChainFuzz $_{expl}$ could only generate PoCs for vulnerabilities CVE-2023-49468 and CVE-2024-31619. Conversely, ChainFuzz $_{exp}$ could generate PoCs for all the vulnerabilities but required 30% more time on average compared to ChainFuzz.

Initial seeds significantly impact the effectiveness of fuzzing approaches [19, 33, 37, 38]. In Section 4.1, we evaluated ChainFuzz using the upstream PoC and one valid downstream input as initial seeds, successfully generating downstream PoCs in all cases. To assess the impact of initial seeds, we conducted an experiment where ChainFuzz was initialized with only the upstream PoC and tasked with generating PoCs for Downstream#2 for each vulnerability listed in Table 1. We replaced each fuzzing procedure five times to ensure consistency. The results show that ChainFuzz successfully generated PoCs for four vulnerabilities, while for six vulnerabilities, it reached the vulnerable code without triggering the vulnerability. The primary challenge is the format incompatibility between upstream and downstream software. For instance, when fuzzing *MuPDF* to generate PoCs for the vulnerability CVE-2023-39328 in *OpenJPEG*, ChainFuzz struggled to generate a valid PDF file embedding the JPEG-2000 PoC. Using proper downstream inputs as seeds, which developers can easily collect, addresses this limitation.

4.2.1 Effectivenes of waypoints

In this experiment, we investigated whether the waypoint-guided cross-layer fuzzing strategy in ChainFuzz helps generate more inputs that can reach waypoints and buggy locations. As an example, we considered CVE-2021-29390, an upstream vulnerability in *libjpeg-turbo* exploitable through *libtiff*. We analyzed the execution traces of inputs generated by different tools within a 24-hour period while testing *libtiff* for CVE-2021-29390. By examining the execution trace of each input, we determined whether it could reach the identified waypoints or the vulnerable function (VF). The waypoints were identified by ChainFuzz. The results, presented in Table 4, show that ChainFuzz generated the highest number of inputs capable of reaching both the waypoints and the vulnerable function. This result demonstrates the effectiveness of its cross-layer directed fuzzing and trace-differential-guided mutation strategies. In contrast, other tools such as AFLGo-Down could generate inputs that reached the vulnerable function but failed to trigger the vulnerability. This limitation arises because these tools cannot satisfy the specific conditions required to exploit the vulnerability.

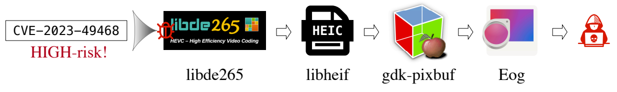


Figure 9: The propagation of CVE-2023-49468.

4.3 Case Studies

Case Study I: Adapting Long Supply Chain. *libde265* is a popular open-source implementation of the H.265 video codec and is integrated into many applications. Eye Of GNOME (Eog) is an image viewer that supports various image formats. It relies on libde265 to parse H.265 data, as shown in Figure 9. Specifically, Eog directly depends on *gdk-pixbuf*, which in turn depends on *libheif*, which ultimately relies on *libde265*. CVE-2023-49468 [29] is a high-risk *Out-of-bounds Write* vulnerability in *libde265*. However, the original PoC for this vulnerability cannot be correctly handled by Eog and therefore fails to trigger the vulnerability. Using this PoC as a reference, ChainFuzz successfully generates a PoC that is usable in Eog to trigger the vulnerability.

Through manual analysis of the vulnerability-triggering path from Eog to *libde265*, we identify significant deviations from the original execution path, resulting in a highly complex and challenging scenario for existing tools to detect. Furthermore, since Eog is desktop software, it only needs to open the PoC generated by ChainFuzz to trigger the upstream vulnerability. This not only simplifies the exploitation process but also broadens the potential impact scope of the vulnerability.

Case Study II: Discovering a New Upstream Vulnerability. CVE-2023-49464 is a high-risk vulnerability in *libheif*, which is relied upon by *OpenImageIO*. Using ChainFuzz, we successfully generate a downstream PoC for *OpenImageIO* that triggers a different but similar upstream vulnerability, as illustrated in Figure 10. The original vulnerability occurs at line 534 in Figure 10(a) and is caused by a heap buffer over-

```
/* Software: libheif, File: libheif/uncompressed_image.cc */
520 int get_luma_bits_pre_pixel(HeifFile& file, heif_item_id imageID) {
        ...
532     for (Box_uncC::Component comp : uncc_box->get_components()){
533         uint16_t comp_index = comp.comp_index;
534         auto comp_type = cmpd_box->get_components()[comp_index].comp_type;
            ...                             Heap buffer overflow
548     }
560 }
```

(a) Buggy site of CVE-2023-49464

```
/* Software: libheif, File: libheif/uncompressed_image.cc */
363 Error uncompressed_image_type_is_support(Box_uncC& uncC, Box_cmpd& cmpd) {
364     for (Box_uncC::Component comp : uncc_box->get_components()){
365         uint16_t comp_index = comp.comp_index;
366         auto comp_type = cmpd_box->get_components()[comp_index].comp_type;
            ...                            Heap buffer overflow
396     }
456 }
```

(b) Buggy site triggered by OpenImageIO

Figure 10: The downstream triggers a different upstream vulnerability.

flow due to an out-of-bounds array index. Using the original PoC, ChainFuzz generates a new PoC for *OpenImageIO* that triggers a different vulnerability in *libheif* at line 366 in Figure 10(b). Although these vulnerabilities occur in different locations, they originate from similar code logic.

This happens because *OpenImageIO* does not invoke the original vulnerable function in *libheif*. Instead, it triggers a vulnerability in another function that contains similar flawed code. As a result, relying solely on analyzing the reachability of upstream vulnerable functions in downstream software may fail to detect such vulnerabilities, leading to false negatives. ChainFuzz mitigates this issue by generating PoCs capable of uncovering such hidden vulnerabilities.

Case Study III: Ineffective Updating of Vulnerable Dependencies. CVE-2023-0802 is a vulnerability in *libtiff*, which is used by *ImageMagick* to handle TIFF files. During testing, we discovered that the PoC for CVE-2023-0802 could trigger a new vulnerability within *ImageMagick*. Even after updating *libtiff* to the latest version, the vulnerability persisted. We reported this issue to the *ImageMagick* developers, who provided an initial fix. However, this fix was incomplete, as we identified new vulnerabilities by adjusting runtime parameters. After further reports, the developers implemented additional fixes to address the issue. This case underscores the complexity of managing software dependencies and the limitations of relying solely on updating vulnerable libraries to mitigate upstream threats. Addressing vulnerabilities that span multiple software components introduces additional challenges and requires more comprehensive solutions.

5 Discussion and Limitations

Reliance on Upstream PoCs. ChainFuzz leverages the original upstream PoC to guide input mutation for downstream PoC generation. On the one hand, the original PoC can often be obtained from publicly available vulnerability reports. On the other hand, many approaches have been proposed to reproduce vulnerabilities and generate PoCs for individual projects. We claim that these approaches are orthogonal to ChainFuzz. ChainFuzz's unique contribution lies in its ability to generate downstream PoCs that exploit upstream vulnerabilities, as the original upstream PoC is often insufficient for this purpose. Users can employ existing approaches to reproduce the upstream vulnerability and then employ ChainFuzz to generate downstream PoCs. Therefore, ChainFuzz can be integrated with other tools, such as AFLGo [2]. We consider this as our future work.

```
1 DeferredImgDecoder DeferredImgDecoder::Create(SharedBuffer data, ...) {
2     String type = SnifMimeInternal(data);
3     ImageDecoder meta_decoder = CreateByMimeType(type, data);
4     DeferredImgDecoder decoder(new DeferredImgDecoder(meta_decoder, ...));
5     decoder->SetDataInternal(data, ...); //Pass file data to the decoder
6     return decoder;
7 }
```

(a)

```
1 ImageDecoder CreateByMimeType(String mime_type, SegmentReader data, ...){
2     ImageDecoder decoder;
3     mime_type = mime_type.LowerASCII();
4     if (mime_type == "image/jpeg" || mime_type == "image/pjpeg" ||
5         mime_type == "image/jpg") {
6         ...
7         decoder = JPEGImageDecoder(...); //Decoder for JPEG Image
8     } else if (mime_type == "image/png" || mime_type == "image/x-png" ||
9                mime_type == "image/apng") {
10        ...
11        decoder = PNGImageDecoder(...); //Decoder for PNG Image
12    } else if (mime_type == "image/avif") {
13        ...
14        decoder = AVIFImageDecoder(...); //Decoder for AVIF Image
15    }
16    if (decoder) {
17        decoder->SetData(data, ...); //P
18    }
19    return decoder;
20 }
```

(b)

```
1 bool AVIFImageDecoder::UpdateDemuxer() {
2     ...
3     auto ret = avifDecoderParse(decoder_.get());//Decode AVIF image
4 }                                             //through libavif
```

(c)

Figure 11: The input decoding process of Chromium.

Generalization of Input Division. In this paper, we focus on generating file-based PoCs for upstream vulnerabilities. Kwon et al. [20] found that 70% of PoCs for Common Vulnerabilities and Exposures (CVE) were malicious file types. Our mutation strategy is based on the idea that inputs to downstream software can be categorized by whether they are processed upstream, downstream, or by both. This structure is a common pattern in software supply chains, where downstream software relies on upstream components for specific tasks, such as file decoding. This improves development efficiency, as downstream developers can focus on understanding and invoking the APIs of upstream libraries to reuse code.

Figure 11 illustrates this concept with a simplified code snippet from Chromium's image decoding module, which supports three file formats. For an AVIF file, the decoder is initialized at line 14 in Figure 11(b), image data is passed to the decoder at line 5 in Figure 11(a), and the data is processed by *libavif* via its API at line 3 in Figure 11(c). Malformed data passed to *libavif* can trigger vulnerabilities, such as CVE-2023-6704. Similar processes are observed in other feature-rich software, like ImageMagick and GNOME.

6 Related Work

Reachability Analysis of Upstream Vulnerabilities. Several methods have been proposed to detect vulnerabilities in reused open-source software (OSS) components. OCTOPOCS [20] introduces an approach to validate the exploitability of propagated vulnerabilities with a reformed PoC, combining taint analysis and directed symbolic execution. Its primary focus is on pairs where the propagated software T directly clones the vulnerable software S. However, in the context of the software supply chain, downstream software typically links dynamically to upstream libraries and involves numerous transitive dependencies. V1SCAN [46] focuses on discovering propagated 1-day vulnerabilities in reused C/C++ OSS components. SIEGE [17] proposes a search-based automatic exploit generation technique to reach vulnerable code within dependencies for Java clients. TRANSFER [18] can also generate test cases for Java projects to demonstrate the exploitability of library vulnerabilities without domain knowledge. `1dFuzz` [48] focuses on reproducing 1-day vulnerabilities by studying security patch characteristics and proposing a directed differential testing technique. Compared to these approaches, CHAINFUZZ is uniquely effective in generating PoCs to validate the exploitability of upstream vulnerabilities, even in complex and long software supply chains.

Directed Greybox Fuzzing. DGF was first introduced by Böhme *et al.* in 2017 [2], and subsequent research has continuously refined and expanded it from various perspectives. DSFuzz [23] aims to reach deep program states, while Titan [16] improves DGF to efficiently reach multiple targets. PDGF [49] presents a predecessor-aware directed DGF method to improve its efficiency. In addition, DGF has been adapted to specific scenarios. For example, SYZDIRECT [43] proposes a DGF framework for the Linux kernel. However, most existing approaches focus primarily on reaching targets within individual projects and are not well suited to assessing the exploitability of upstream vulnerabilities in software supply chain scenarios. CHAINFUZZ addresses this limitation by identifying *waypoints* as intermediate targets and using seed prioritization and mutation strategies to effectively validate upstream vulnerabilities.

7 Conclusion

In this paper, we proposed CHAINFUZZ, a novel approach to validate the exploitability of upstream vulnerabilities by generating PoCs for downstream software. We thoroughly evaluated CHAINFUZZ on a comprehensive dataset, demonstrating its effectiveness in generating PoCs for vulnerabilities in both direct and transitive dependencies. Our results show that CHAINFUZZ outperforms state-of-the-art approaches, addressing the limitations of existing methods and providing a robust solution for software supply chain security.

Acknowledgments

We would like to thank anonymous shepherd and reviewers for their helpful comments and feedback. This work was supported in part by the National Natural Science Foundation of China (62102093, U2436207, 62172104, 62172105, 62202106, 62302101, 62102091, 62472096, 62402114, 62402116). Min Yang is the corresponding author and a faculty of Shanghai Institute of Intelligent Electronics & Systems, and Engineering Research Center of Cyber Security Auditing and Monitoring, Ministry of Education, China. Yuan Zhang was supported in part by the Shanghai Pilot Program for Basic Research - FuDan University 21TQ1400100 (21TQ012).

Ethics considerations

We have disclosed all identified zero-day vulnerabilities to the respective manufacturers and assisted in remediation.

Open science

We have made CHAINFUZZ and our datasets publicly available at https://zenodo.org/records/14732712.

References

[1] CVE-2021-29390. `https://nvd.nist.gov/vuln/detail/CVE-2021-29390`.

[2] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 2329–2344, 2017.

[3] Nicholas Boucher and Ross Anderson. Trojan source: Invisible vulnerabilities. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6507–6524, 2023.

[4] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pages 2095–2108, 2018.

[5] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725. IEEE, 2018.

[6] Peng Chen, Jianzhong Liu, and Hao Chen. Matryoshka: fuzzing deeply nested branches. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 499–513, 2019.

[7] Jiarun Dai, Yuan Zhang, Hailong Xu, Haiming Lyu, Zicheng Wu, Xinyu Xing, and Min Yang. Facilitating vulnerability assessment through poc migration. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 3300–3317, 2021.

[8] Peng Deng, Zhemin Yang, Lei Zhang, Guangliang Yang, Wenzheng Hong, Yuan Zhang, and Min Yang. Nestfuzz: Enhancing fuzzing with comprehensive understanding of input processing logic. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 1272–1286, 2023.

[9] Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. Keep me updated: An empirical study of third-party library updatability on android. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2187–2200, 2017.

[10] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. Lava: Large-scale automated vulnerability addition. In *2016 IEEE symposium on security and privacy (SP)*, pages 110–121. IEEE, 2016.

[11] William Enck and Laurie Williams. Top five challenges in software supply chain security: Observations from 30 industry and government organizations. *IEEE Security & Privacy*, 20(2):96–100, 2022.

[12] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. {AFL++}: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.

[13] Darius Foo, Jason Yeo, Hao Xiao, and Asankhaya Sharma. The dynamics of software composition analysis. *arXiv preprint arXiv:1909.00973*, 2019.

[14] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. Magma: A ground-truth fuzzing benchmark. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 4(3):1–29, 2020.

[15] Heqing Huang, Yiyuan Guo, Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. Beacon: Directed grey-box fuzzing with provable path pruning. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 36–50. IEEE, 2022.

[16] Heqing Huang, Peisen Yao, Hung-Chun Chiu, Yiyuan Guo, and Charles Zhang. Titan: Efficient multi-target directed greybox fuzzing. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 59–59. IEEE Computer Society, 2023.

[17] Emanuele Iannone, Dario Di Nucci, Antonino Sabetta, and Andrea De Lucia. Toward automated exploit generation for known vulnerabilities in open-source libraries. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pages 396–400. IEEE, 2021.

[18] Hong Jin Kang, Truong Giang Nguyen, Bach Le, Corina S Păsăreanu, and David Lo. Test mimicry to assess the exploitability of library vulnerabilities. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 276–288, 2022.

[19] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pages 2123–2138, 2018.

[20] Seongkyeong Kwon, Seunghoon Woo, Gangmo Seong, and Heejo 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)*, pages 174–185. IEEE, 2021.

[21] Tobias Lauinger, Abdelberi Chaabane, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web. *arXiv preprint arXiv:1811.00918*, 2018.

[22] Gwangmu Lee, Woochul Shim, and Byoungyoung Lee. Constraint-guided directed greybox fuzzing. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3559–3576, 2021.

[23] Yinxi Liu and Wei Meng. Dsfuzz: Detecting deep state bugs with dependent state exploration. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 1242–1256, 2023.

[24] LLVM. DataFlowSanitizer. `https://clang.llvm.org/docs/DataFlowSanitizer.html`, 2014.

[25] Shane McIntosh, Bram Adams, Thanh HD Nguyen, Yasutaka Kamei, and Ahmed E Hassan. An empirical study of build maintenance effort. In *Proceedings of the 33rd international conference on software engineering*, pages 141–150, 2011.

[26] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. Fuzzbench: an open fuzzer benchmarking platform and service. In *Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, pages 1393–1403, 2021.

[27] Samim Mirhosseini and Chris Parnin. Can automated pull requests encourage software developers to upgrade out-of-date dependencies? In *2017 32nd IEEE/ACM international conference on automated software engineering (ASE)*, pages 84–94. IEEE, 2017.

[28] Dongliang Mu, Alejandro Cuevas, Limin Yang, Hang Hu, Xinyu Xing, Bing Mao, and Gang Wang. Understanding the reproducibility of crowd-reported security vulnerabilities. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 919–936, 2018.

[29] NVD. CVE-2023-49468. `https://nvd.nist.gov/vuln/detail/CVE-2023-49468`, 2023.

[30] NVD. CVE-2024-3094. `https://nvd.nist.gov/vuln/detail/CVE-2024-3094`, 2024.

[31] NVD. Vulnerabilities. `https://nvd.nist.gov/vuln`, 2024.

[32] OWASP. Dependency-Check. `https://owasp.org/www-project-dependency-check/`, 2024.

[33] Shankara Pailoor, Andrew Aday, and Suman Jana. {MoonShine}: Optimizing {OS} fuzzer seed selection with trace distillation. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 729–743, 2018.

[34] Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. Vulnerable open source dependencies: Counting those that matter. In *Proceedings of the 12th ACM/IEEE international symposium on empirical software engineering and measurement*, pages 1–10, 2018.

[35] Ivan Pashchenko, Duc-Ly Vu, and Fabio Massacci. A qualitative study of dependency management and its security implications. In *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*, pages 1513–1531, 2020.

[36] Serena Elisa Ponta, Henrik Plate, and Antonino Sabetta. Beyond metadata: Code-centric and usage-based analysis of known vulnerabilities in open-source software. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 449–460. IEEE, 2018.

[37] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. Optimizing seed selection for fuzzing. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 861–875, 2014.

[38] Moritz Schloegel, Nils Bars, Nico Schiller, Lukas Bernhard, Tobias Scharnowski, Addison Crump, Arash Ale-Ebrahim, Nicolai Bissantz, Marius Muench, and Thorsten Holz. Sok: Prudent evaluation practices for fuzzing. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 1974–1993. IEEE, 2024.

[39] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. {AddressSanitizer}: A fast address sanity checker. In *2012 USENIX annual technical conference (USENIX ATC 12)*, pages 309–318, 2012.

[40] Abhishek Shah, Dongdong She, Samanway Sadhu, Krish Singal, Peter Coffman, and Suman Jana. Mc2: Rigorous and efficient directed greybox fuzzing. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2595–2609, 2022.

[41] snyk. Snyk. `https://snyk.io/`, 2024.

[42] Yulei Sui and Jingling Xue. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th international conference on compiler construction*, pages 265–266. ACM, 2016.

[43] Xin Tan, Yuan Zhang, Jiadong Lu, Xin Xiong, Zhuang Liu, and Min Yang. Syzdirect: Directed greybox fuzzing for linux kernel. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 1630–1644, 2023.

[44] Wei Tang, Zhengzi Xu, Chengwei Liu, Jiahui Wu, Shouguo Yang, Yi Li, Ping Luo, and Yang Liu. Towards understanding third-party library dependency in c/c++ ecosystem. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–12, 2022.

[45] wikipedia. Cohen's kappa wikipedia. `https://en.wikipedia.org/wiki/Cohen%27s_kappa`.

[46] Seunghoon Woo, Eunjin Choi, Heejo Lee, and Hakjoo Oh. {V1SCAN}: Discovering 1-day vulnerabilities in reused {C/C++} open-source software components using code classification techniques. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6541–6556, 2023.

[47] Yulun Wu, Zeliang Yu, Ming Wen, Qiang Li, Deqing Zou, and Hai Jin. Understanding the threats of upstream vulnerabilities to downstream projects in the maven ecosystem. in 2023 ieee/acm 45th international conference on software engineering (icse), 2023.

[48] Songtao Yang, Yubo He, Kaixiang Chen, Zheyu Ma, Xiapu Luo, Yong Xie, Jianjun Chen, and Chao Zhang. 1dfuzz: Reproduce 1-day vulnerabilities with directed differential fuzzing. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 867–879, 2023.

[49] Yujian Zhang, Yaokun Liu, Jinyu Xu, and Yanhao Wang. Predecessor-aware directed greybox fuzzing. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 40–40. IEEE Computer Society, 2023.

[50] Peiyuan Zong, Tao Lv, Dawei Wang, Zizhuang Deng, Ruigang Liang, and Kai Chen. {FuzzGuard}: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning. In *29th USENIX security symposium (USENIX security 20)*, pages 2255–2269, 2020.

Appendices

Table 5: Details of the exploitable vulnerabilities in our dataset. "Exploitability" refers to whether the vulnerability can be exploited in the downstream software. The vulnerabilities of Case#2 are further evaluated in Table 1.

ID	Software	Version	Vulnerability	Vulnerability Type	Cmd	Exploitability
1	libheif	1.17.5	CVE-2023-49460	Segment Fault	./heif-convert poc test.png	Case1
2	libheif	1.17.5	CVE-2023-49462	Segment Fault	./heif-convert poc test.png	
3	libheif	1.17.5	CVE-2023-49463	Segment Fault	./heif-convert poc test.png	
4	libheif	1.17.5	CVE-2023-49464	Use After Free	./heif-convert poc test.png	
5	libheif	1.17.6	CVE-2024-31619	Segment Fault	./heif-info poc	
6	libtiff	4.3.0	CVE-2022-0561	NULL Pointer Dereference	./tiffinfo -f lsb2msb -Dcdjrsz poc	
7	libtiff	4.3.0	CVE-2022-0562	NULL Pointer Dereference	./tiffinfo -f lsb2msb -Dcdjrsz poc	
8	libtiff	4.3.0	CVE-2022-0908	Null Pointer Dereference	./tiff2pdf poc	
9	libde265	1.0.4	CVE-2020-21600	heap buffer overflow	./dec265 poc	Case2
10	libde265	1.0.8	CVE-2022-43239	heap buffer overflow	./dec265 poc	
11	libde265	1.0.8	CVE-2022-43252	heap buffer overflow	./dec265 poc	
12	libde265	1.0.10	CVE-2023-24751	NULL Pointer Dereference	./dec265 poc	
13	libde265	1.0.10	CVE-2023-24752	NULL Pointer Dereference	./dec265 poc	
14	libde265	1.0.10	CVE-2023-25221	heap buffer overflow	./dec265 poc	
15	libde265	1.0.14	CVE-2023-49468	global buffer overflow	./dec265 poc	
16	libheif	1.17.6	CVE-2024-31619	Segment Fault	./heif-info poc	
17	libjpeg-turbo	2.0.90	CVE-2021-29390	heap buffer overflow	./djpeg poc.jpg	
18	OpenJPEG	2.3.0	CVE-2019-6988	excessive memory allocation	./opj_decompress -i poc -o out.png	
19	OpenJPEG	2.1.1	CVE-2016-10506	Divide By Zero	./opj_decompress -i poc -o out.pgm	
20	OpenJPEG	2.5.0	CVE-2023-39328	Uncontrolled Resource Consumption	./opj_decompress -i poc -o te.raw	
21	OpenJPEG	2.1.1	Issue#399	heap buffer overflow	./opj_decompress -i poc -o out.pgm	
22	OpenJPEG	2.1.1	Issue#394	heap buffer overflow	./opj_decompress -i poc -o out.pgm	
23	OpenJPEG	2.1.1	Issue#389	heap buffer overflow	./opj_decompress -i poc -o out.pgm	
24	OpenJPEG	2.1.1	Issue#393	heap double free	./opj_decompress -i poc -o out.pgm	
25	OpenJPEG	2.5.0	Issue#1501	SIGILL	./opj_decompress -i poc -o out.pgm	
26	OpenJPEG	2.5.0	Issue#1505	NULL Pointer Dereference	./opj_decompress -i poc -r 5 -o out.ppm	
27[1]	libtiff	4.3.0	CVE-2022-0561	NULL Pointer Dereference	./tiffinfo -f lsb2msb -Dcdjrsz poc	
28[1]	libtiff	4.3.0	CVE-2022-0562	NULL Pointer Dereference	./tiffinfo -f lsb2msb -Dcdjrsz poc	
29[1]	libtiff	4.3.0	CVE-2022-0908	Null Pointer Dereference	./tiff2pdf poc	

[1] The PoCs for these vulnerabilities can directly trigger the same issue through ImageMagick, libwebp, and OpenJPEG. However, when used with PoDoFo, mutations are required for successful exploitation.

Table 6: Details of the unexploitable vulnerabilities in our dataset.

Upstream	Version	Vulnerability	Vulnerability Type	Cmd	Downstream	Exploitability
libxml2	2.9.10	CVE-2020-24977	global buffer over-read	./xmllint --htmlout poc	Podofo	unreachable
libxml2	2.9.11	CVE-2021-3516	use after free	./xmllint --nocompact --html --push poc	Podofo	unreachable
libxml2	2.9.11	CVE-2021-3517	out-of-bounds write	./xmllint --recover --postvalid poc	Podofo	unreachable
libxml2	2.9.11	CVE-2021-3518	use after free	./xmllint --recover --dropdtd --nofixup-base-uris poc	Podofo	unreachable
libxml2	2.9.11	CVE-2021-3537	Null Pointer Dereference	./xmllint --recover --postvalid poc	Podofo	unreachable
LibRaw	0.21.1	CVE-2023-1729	heap buffer overflow	./dcraw_half poc	OpenImageIO	exploitable
LibRaw	0.21.3	Issue#400	stack buffer overflow	./libraw_cr2_fuzzer poc	OpenImageIO	reachable
libpng	1.6.34	CVE-2018-13785	Divide By Zero	./pngimage poc	libgd	unreachable
libpng	1.6.34	CVE-2018-14048	heap buffer overflow	./blackwhite poc	libgd	unreachable
libpng	1.6.37	CVE-2019-7317	use after free	crafted driver	libgd	reachable
libpng	1.6	CVE-2021-4214	buffer overflow	./pngimage poc	libgd	unreachable
libpng	1.6.37	CVE-2019-14373	heap buffer overflow	crafted driver	libgd	unreachable
freetype	53dfdcd8	CVE-2022-27405	heap buffer overflow	crafted driver	MuPDF	reachable
freetype	22a0cccb	CVE-2022-27406	heap buffer overflow	crafted driver	MuPDF	reachable
freetype	2.12.1	CVE-2022-31782	heap buffer overflow	./ftbench -c 1 poc	MuPDF	unreachable
OpenJPEG	2.3.0	CVE-2018-18088	Null Pointer Dereference	./opj_decompress -i poc -o out.ppm	MuPDF/libheif	unreachable
OpenJPEG	2.3.1	CVE-2020-6851	heap buffer overflow	./opj_decompress -i poc -o out.pgm	MuPDF/libheif	reachable
OpenJPEG	2.4.0	CVE-2020-27814	heap buffer overflow	./opj_compress -i poc -o out.j2k -M 3	MuPDF/libheif	unreachable
OpenJPEG	2.4.0	CVE-2021-3575	heap buffer overflow	./opj_decompress -i poc -o out.png	MuPDF/libheif	unreachable
OpenJPEG	2.4.0	CVE-2022-1122	heap buffer overflow	./opj_decompress -ImgDir poc -OutFor BMP	MuPDF/libheif	unreachable
libde265	1.0.10	CVE-2023-24756	Null Pointer Dereference	./dec265 poc	libheif	reachable
libde265	1.0.10	CVE-2023-24757	Null Pointer Dereference	./dec265 poc	libheif	reachable
libde265	1.0.10	CVE-2023-24758	Null Pointer Dereference	./dec265 poc	libheif	reachable
libde265	1.0.10	CVE-2023-25221	heap buffer overflow	./dec265 poc	libheif	reachable
libde265	1.0.11	CVE-2023-27102	heap buffer overflow	./dec265 poc	libheif	reachable
libjpeg	2.0.3	CVE-2020-17541	stack buffer overflow	./jpegtran-static -transverse poc	libtiff	unreachable
libjpeg	2.0.5	CVE-2020-35538	Null Pointer Dereference	./djpeg -fast -skip 1,20 -outfile out poc	libtiff	reachable
libjpeg	2.0.91	CVE-2021-20205	Divide By Zero	./cjpeg poc	libtiff	unreachable
libjpeg	84d6306f	CVE-2021-37972	out-of-bounds read	./jpegtran -outfile x poc	libtiff	unreachable
libjpeg	2.1.92	CVE-2023-2804	heap buffer overflow	./djpeg -nosmooth poc	libtiff	unreachable
libtiff	b51bb157	CVE-2022-1622	out-of-bounds read	./tiffcp -i poc /tmp/foo	OpenJPEG/ImageMagick	reachable
libtiff	b51bb157	CVE-2022-1623	out-of-bounds read	./tiffcp -i poc /tmp/foo	OpenJPEG/ImageMagick	reachable
libtiff	4.5.0	CVE-2023-2731	Null Pointer Dereference	./tiffcp -i poc /dev/null	OpenJPEG/ImageMagick	reachable
libtiff	4.6.0	CVE-2023-52355	out-of-bounds write	crafted driver	OpenJPEG/ImageMagick	unreachable
libtiff	4.6.0	CVE-2023-52356	heap buffer overflow	crafted driver	OpenJPEG/ImageMagick	unreachable