SPFUZZ: Validating Secretly Patched Vulnerabilities by Patch Guided Fuzzing

Shiyan Peng
pengsy21@m.fudan.edu.cn
Fudan University
Shanghai, China

Yingbo Xu
23210240347@m.fudan.edu.cn
Fudan University
Shanghai, China

Jiarun Dai
jrdai@fudan.edu.cn
Fudan University
Shanghai, China

Yuchen Zhang
yuchenzhang@bnbu.edu.cn
Beijing Normal–Hong Kong Baptist University
Zhuhai, China

Yuan Zhang
yuanxzhang@fudan.edu.cn
Fudan University
Shanghai, China

Min Yang
m_yang@fudan.edu.cn
Fudan University
Shanghai, China

Abstract

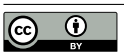
Secretly patched vulnerabilities (i.e., vulnerabilities fixed without public disclosure) are common in open-source software and can be easily overlooked during downstream security maintenance. Existing approaches mainly rely on static analysis or machine learning to identify candidate security patches. However, they do not directly determine whether such patch candidates actually correspond to reachable vulnerability-triggering executions in the pre-patch program.

This paper presents SPFuzz, a patch-guided directed fuzzing framework for detecting secretly patched vulnerabilities through proof-of-concept (PoC) generation. Rather than classifying or validating patches themselves, SPFuzz takes a candidate security patch as input and uses it to attempt to trigger the underlying pre-patch vulnerability, thereby providing direct runtime evidence of the hidden risk. SPFuzz derives Vulnerability Triggering States (VTS), a patch-derived guidance signal that steers fuzzing toward vulnerability-triggering executions. SPFuzz further supports coordinated two-dimensional fuzzing across user inputs and command-line options, enhancing the effectiveness of PoC generation and enabling a more direct assessment of secretly patched vulnerabilities from candidate security patches.

We evaluate SPFuzz on two datasets. On 34 recent CVE patches, SPFuzz successfully generated 18 PoCs, drastically outperforming option-aware fuzzers (18 vs. 7 cases). SPFuzz also outperforms state-of-the-art directed fuzzers when only the correct options and patch sites are provided. Furthermore, in a vulnerability discovery experiment on 110 candidate security patches, SPFuzz detected 17 latent vulnerabilities across seven well-tested open-source projects, yielding 4 new CVEs.

CCS Concepts

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



This work is licensed under a Creative Commons Attribution 4.0 International License.
CCS '26, The Hague, Netherlands
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2871-6/2026/11
https://doi.org/10.1145/3830454.3846746

Keywords

Fuzzing; Secret Patch

ACM Reference Format:
Shiyan Peng, Yingbo Xu, Jiarun Dai, Yuchen Zhang, Yuan Zhang, and Min Yang. 2026. SPFUZZ: Validating Secretly Patched Vulnerabilities by Patch Guided Fuzzing. In *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS '26), November 15–19, 2026, The Hague, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3830454.3846746

1 Introduction

Although open-source software (OSS) has significantly advanced software development, vulnerabilities in OSS routinely propagate to downstream software and create supply-chain security risks. In practice, downstream maintainers cannot promptly audit every upstream commit in time and therefore depend heavily on explicit vulnerability disclosures to prioritize patch adoption. When security fixes are silently merged without such disclosures, downstream users are left with patched code but without knowing the actual threats of the vulnerability. As a result, they may fail to prioritize patch adoption or fail to recognize that deployed versions remain exposed to the underlying vulnerability. Recent studies [25, 31] have shown that such secretly patched vulnerabilities are widespread, leaving software unknowingly exposed to hidden security risks.

To tackle this type of vulnerability, the security community suggests comprehensively identifying security patches in OSS. Technically, secret patch identification is typically achieved via machine learning models [29, 30, 32, 37–39] utilizing token and structural features, or static analysis approaches [19, 22, 24, 34] based on patch pattern matching. However, the lack of runtime context in them makes false positives unavoidable. While their output could tell if a candidate security patches which patches are security related, they are limited to identifying confirming and provide acual information.

Considering the above, we aim to detect secretly patched vulnerabilities through automated PoC generation on the pre-patch program, using a candidate security patch as input. Specifically, we construct program inputs that reproduce the vulnerability fixed by the patch. This detection solution is for further exposing vulnerability instead of on classify or validate the patch. It uses the patch input to discover and confirm the underlying vulnerability with concrete runtime evidence. These PoCs provide the verifiable proof

absent in classification-based methods because they confirm both the presence of the vulnerability and its practical impact.

Literature Review. Directed greybox fuzzing (DGF) is an established technique for generating PoCs of specific program targets. However, we observe that the existing DGF methods face two key limitations when applied to the detection of secretly patched vulnerabilities: ❶ *Unsuitable for Pre-patch Code Fuzzing.* In pre-patch testing, merely reaching the patch location is not enough: the fuzzer must steer execution toward the vulnerability-triggering state that the patch was meant to prevent. So general DGF techniques [8, 9, 13, 18] exhibit significant limitations. These methods primarily optimize for code reachability, directing mutation resources merely toward reaching the patch locations rather than establishing the specific states necessary to trigger the underlying vulnerability. Patch-oriented DGFs [35, 40] extend the target scope to include any code maintaining control- or data-flow dependencies with the patch. This strategy indiscriminately pulls in vast amounts of benign logic (e.g., logging or utilities) that are irrelevant to the vulnerability fixed by the patch. Consequently, the fuzzer wastes resources on these unrelated paths rather than focusing on the vulnerability. Other DGF approaches [7, 10, 14, 16, 17] rely on explicit failure indicators (e.g., stack traces from crash dumps) that are unavailable in a patch-only context. Consequently, they are fundamentally inapplicable to pre-patch testing. ❷ *Neglect Command-Line Options.* Triggering real-world vulnerabilities may depend not only on user inputs but also on command-line option combinations. Therefore, detecting them requires attempting various combinations of options. This is a difficult task since modern programs support a large number of options. Current DGF approaches overlook this problem, leading to limited effectiveness in identifying secretly patched vulnerabilities. Meanwhile, option-aware fuzzers such as CarpetFuzz [26], ProphetFuzz [27], OSmart [28], and ZigZagFuzz [15] mainly optimize coverage over the option space; they are not designed to use patch semantics to determine which options are relevant for exposing a specific pre-patch vulnerability.

Challenges. To address the aforementioned limitations, we propose a novel patch-oriented directed greybox fuzzer capable of jointly exploring both user inputs and command-line options in the pre-patch code. However, building such a fuzzer is non-trivial due to the following challenges.

- **Challenge-I: How to formalize a patch into effective pre-patch code fuzzing feedback?** A patch merely indicates how a target vulnerability is guarded; it does not directly provide the precise runtime program states required to trigger the underlying vulnerability. Hence, curating reliable fuzzing guidance for detecting secretly patched vulnerabilities in the pre-patch code is challenging. Moreover, patches do not follow a standard pattern and can appear in many different forms. This variety makes it difficult for tools to extract useful feedback for fuzzing.
- **Challenge-II: How to jointly fuzz user inputs and option combinations?** Modern programs typically expose a rich configuration option space, where only a small subset of option combinations can activate the vulnerable functionality. As a result, the combined space of options and inputs grows exponentially, rendering brute-force exploration infeasible. Meanwhile, patches do not contain explicit information about options, making it difficult to infer which options are necessary to reach the patch.

Our Solutions. To address Challenge-I, we develop a patch-to-feedback transformation system centered on Vulnerability-Triggering States (VTS), which provide actionable guidance even for patches without explicit vulnerability descriptions. VTS quantify the satisfaction of vulnerability conditions through patch-related variables, enabling the fuzzer to track the triggering process and optimize power scheduling. Inspired by prior patch-behavior studies [22, 24, 33], we extend their classification by incorporating variable type information, designing a taxonomy-driven transformation that maps each patch category to behavior- and type-specific inference rules. This ensures robust handling of diverse real-world patches while generating effective guidance for pre-patch fuzzing.

To address Challenge-II, we propose a VTS-guided two dimensional fuzzing strategy. The fuzzing process jointly allocates energy and prioritizes inputs across both user inputs and command-line options, enabling the fuzzer to progressively satisfy vulnerability conditions and effectively trigger the pre-patch vulnerability. By observing how mutations in either dimension influence the patch-derived state, our approach can assess semantic progress toward vulnerable behaviors without requiring explicit knowledge of option dependencies. This feedback enables coordinated scheduling in both option and user input fuzzing, prioritizes mutation results that activate relevant functionality, and effectively prunes the vast joint search space. As a result, our solution achieves efficient two-dimensional directed fuzzing for generating vulnerability-triggering PoC inputs.

Key Results. Following these insights, we implement SPFuzz (Secret Patch Fuzz), a patch-oriented directed greybox fuzzing framework enhanced with vulnerability-triggering state guidance. We conduct experiments to evaluate its effectiveness and practicality on real-world vulnerabilities. We use two datasets with different orientations. The first is a PoC-oriented CVE benchmark of recent cases with public patches and public PoCs; it highlights the necessity of explicitly considering command-line options during fuzzing and demonstrates that SPFuzz can reproduce real-world vulnerabilities under this two-dimensional setting (18/34 cases without predefined correct options). The second is a real-world 110 candidate security patch from recent security-fix commits of seven well-tested open-source projects, where the commits serve as the only available inputs for uncovering the underlying vulnerabilities. SPFuzz successfully generates 17 newly found proof-of-concept inputs, and 4 are subsequently assigned CVE IDs.

Contributions. In summary, this paper makes the following contributions:

- We propose SPFuzz, a patch-guided and option-aware directed fuzzing approach that uses candidate security patches to generate PoC triggers for secretly patched vulnerabilities.
- We introduce Vulnerability-Triggering States (VTS), a novel feedback mechanism that enables joint optimization of user inputs and command-line options during the fuzzing process. Furthermore, we design a patch-to-VTS inference system that can be applied to diverse patch types.
- We conduct comprehensive experiments to evaluate the effectiveness and practicality of SPFuzz, showing strong improvements

over representative directed fuzzing baselines and demonstrating that candidate security patches can be turned into concrete evidence of secretly patched vulnerabilities.

2 Background

2.1 Secretly Patched Vulnerabilities

With the widespread adoption of open-source software (OSS), vulnerabilities in OSS have become a major security concern. Although timely patching is an effective defense against N-day attacks, users and system administrators are often overwhelmed by the growing volume of software updates and must decide which patches to prioritize. In practice, this prioritization often depends on whether a patch is clearly identified as security-relevant.However, many security fixes are merged or released without explicit disclosure in vulnerability databases.We refer to the vulnerabilities fixed but not disclosed as secretly patched vulnerabilities. As a result, downstream users may overlook security-critical updates and remain exposed to the underlying risk.

To detect this type of vulnerability, existing approaches largely rely on static program analysis or machine learning solutions [25, 29, 31, 39], which often fail to accurately assess exploitability and tend to produce ambiguous or overly conservative conclusions. In contrast, a concrete proof-of-concept (PoC) that demonstrates how a vulnerability can be triggered in pre-patch code provides a much more direct and reliable indication of real-world security risk, making PoC generation especially valuable for analyzing these vulnerabilities. In practice, generating such PoCs is challenging not only because the patch must be interpreted as a fuzzing target, but also because many real-world vulnerabilities are exposed only under specific option configurations, such as non-default command-line options, which should be explored in PoC generation.

2.2 Preliminary Study of Vulnerability Triggering Options Requirement

To empirically prove the requirement of non-default command-line options, we conducted a preliminary study demonstrating how often specific option combinations are required to trigger vulnerabilities. We fixed six widely used open-source projects as the study scope: `ffmpeg`, `binutils`, `libavif`, `llama.cpp`, `libtiff`, and `libxml2`. We chose these projects because they are popular CLI utilities with rich functionality, a long history of memory-corruption vulnerabilities and fixes, and sufficient public disclosure material to support case-by-case PoC inspection. We then collected all CVEs for these projects from the NVD that were disclosed between November 2023 and April 2025, yielding 102 CVEs in total. For each CVE, we manually inspected the disclosure issue, commit history, and any public proof-of-concept (PoC) material to determine whether reproducing the vulnerability on the released binary requires non-default command-line options. We conservatively excluded 40 CVEs that lacked public PoCs or sufficiently detailed reproduction information, and retained the remaining 62 cases.

The results are summarized in Table 1. We classify the triggering requirements into *non-default options*, *harness programs*, and *default options*. The results indicate that the majority of PoCs within the PoC-available subset (44/62, 71.0%) rely on non-default option combinations. This study suggests that option combinations play a critical role in vulnerability triggering and motivates us to design a PoC generation that jointly explores both user inputs and options.

Table 1: Statistics of CVE PoCs Categorized by Requirements

Category	Count
Total CVE PoCs audited	62
PoCs with non-default options	44
PoCs with harness program	10
PoCs with default options	8

2.3 Existing PoC Generation Works

Directed greybox fuzzing (DGF) is widely adopted for PoC generation by guiding input mutation toward specific targets. However, detecting secretly patched vulnerabilities fundamentally necessitates two capabilities: guidance tailored for *pre-patch* logic and the exploration of option combinations, both of which are critical for triggering latent vulnerabilities.

Existing work addresses parts of this problem, but not this combination directly. General DGF approaches guide execution toward critical locations [8, 13, 18], and some additionally leverage prior information such as crash reports or static analysis results [7, 10, 16, 17]. However, general DGF methods are insufficient for our scenario, as they focus on reaching target locations without ensuring that the vulnerability-triggering conditions are satisfied, while methods that rely on external artifacts are inapplicable because such materials are not available for detecting previously unknown vulnerabilities. Patch-oriented fuzzers [35, 40] move closer to our setting by treating changed or patch-related code as fuzzing targets. However, target reachability alone is not directly progressing toward the vulnerability-triggering condition the patch implied.

Option-aware fuzzers such as CarpetFuzz [26], ProphetFuzz [27], OSmart [28], and ZigZagFuzz [15] demonstrate that command-line options space is an important source of PoC inputs generation. Their main goal, however, is to improve general exploration over the option space rather than to use patch-derived semantics to decide which option mutations move execution closer to a specific pre-patch vulnerability condition. Thus, they are highly relevant to our scenario, but they do not directly solve the patch-conditioned PoC-generation problem studied here.

In summary, existing techniques face two fundamental limitations: (1) they lack semantic guidance to effectively direct fuzzing toward vulnerable behaviors in the pre-patch version, and (2) they provide no systematic mechanism for exploring user inputs and option combinations in pre-patch code testing scenarios.

3 Challenges and Insights

3.1 Target Scenario

To detect secretly patched vulnerabilities, our objective is to generate a PoC that triggers the vulnerability fixed by a given candidate security patch. In our setting, the patch is an input to the system rather than the object being classified or validated: the actual target is the underlying vulnerability in the pre-patch program. A PoC consists of both user input and a command-line option combination for the released program. Such PoCs let maintainers directly

verify whether the candidate security patch implies a practically risk in their deployed software. Formally, our scenario is defined as follows: *Input:* A candidate security patch and its related program. *Output:* Vulnerability-triggering PoCs that include user inputs and corresponding command-line option combinations. Note that command-line option combinations determine whether a program activates specific functionalities related to the patch, as well as the input sources that can reach the patched code. Therefore, correctly setting these options is also necessary when attempting to expose undisclosed vulnerabilities from a patch.

3.2 Challenges

```
@@ -7112,6 +7118,7 @@ static int loadImage
+    check_buffsize = buffsize + NUM_BUFF_OVERSIZE_BYTES;

     read_buff[buffsize] = 0;
     read_buff[buffsize + 1] = 0;
@@ -7794,6 +7801,11 @@ static int extractImageSection
+    if (src_offset + full_bytes >= check_buffsize)
+    {
+        printf("Bad input. \n");
+        return(-1);
+    }
```

Figure 1: Motivating Example: `1b74e21@libtiff`

Motivating Example. Figure 1 presents a real-world case—commit `1b74e21` from the `libtiff` project—which corresponds to a security patch fixing a heap buffer overflow caused by an out-of-bounds access. To design a fuzzing procedure that can effectively trigger the vulnerability, two requirements must be satisfied. First, beyond merely reaching the patched code, the fuzzing procedure should be able to drive execution to the condition where the access index goes out of bounds. Second, the vulnerability is triggered when the access offset is substantially increased by the rotation functionality within `tiffcrop`. Since this functionality is activated specifically by the `-R` option rather than the user input file alone, successful PoC generation requires a fuzzing procedure capable of exploring the command-line option space. Meeting these two requirements—satisfying the triggering condition and providing the necessary option combination—presents a significant challenge for existing directed fuzzing techniques[7, 8, 10, 13, 14, 16–18, 35, 40].

Challenge-I: How to formalize a patch into effective pre-patch code fuzzing feedback? The effectiveness of fuzzing highly depends on the quality of fuzzing feedback. Directed greybox fuzzing approaches derive high-quality feedback through sophisticated inference from their targets. In our scenario, however, the patch is the only available source of information. Unlike other vulnerability-related sources, a patch rarely provides explicit or richly structured descriptions of the underlying vulnerability, such as crash contexts or root causes. Although the patch location is explicit, coverage of this location is only a necessary condition for PoC generation and is insufficient on its own. Consequently, when testing pre-patch versions, fuzzers lack productive feedback.

This challenge is further exacerbated by the highly diverse forms of vulnerability-fixing logic: patches may manifest as boundary checks, state validations, error handling paths, control-flow restructuring, or subtle updates to data structure invariants. Each form conveys vulnerability-related semantics in a different and implicit manner, which hinders the design of a generic mechanism for systematically interpreting patch semantics and transforming them into fine-grained feedback. Consequently, existing directed fuzzing approaches[7, 8, 10, 13, 14, 16–18, 35, 40] either reduce patches to coarse-grained reaching signals or overlook their implicit semantics, resulting in limited guidance for pre-patch testing.

In the motivating example, the patched code resides in low-level image-processing routines that are frequently executed, making the patch location itself easy to reach during fuzzing. However, merely reaching the location near the patch in the pre-patch code is insufficient to trigger the vulnerable state, which is led by the rotation function. Moreover, even for out-of-bounds vulnerabilities, fixing logic does not follow a single canonical pattern. For example, the patch may be implemented by reassigning the `src_offset` to a safe value. In such cases, no explicit condition predicates are exposed in the patch, unlike in the version shown earlier. This diversity in patch styles further complicates the general extraction of meaningful feedback.

Challenge-II: How to jointly fuzz user inputs and option combinations? Our scenario introduces a new requirement: the fuzzer should also provide the command-line option combination that activates the target functionality, so the patch can be tested.

However, satisfying this requirement is challenging for two reasons. First, the option space is large. Real-world CLI tools expose dozens of options (e.g., tiffcrop with 36+ options) whose combinations produce an enormous search space, far beyond what exhaustive or unguided exploration can cover. Second, patches typically do not contain information about how to reproduce the vulnerability. In particular, they rarely indicate which options are relevant to themselves. As a result, inferring the specific option combinations required to expose the vulnerability becomes a core challenge in the patch-only setting. Existing option-aware fuzzers [15, 26–28] mainly optimize option exploration for coverage growth, but do not provide a patch-conditioned signal for deciding whether an option mutation is moving execution closer to the vulnerable state.

In the motivating example, the vulnerability lies in the rotation functionality. Without setting the correct option `-R`, the program does not drive `src_offset + full_bytes` into an out-of-bounds region, and the vulnerability is not triggered. As a result, existing DGF techniques [7, 8, 10, 13, 14, 16–18, 35, 40] that stay near default option settings miss valid PoCs and fail to fully test the pre-patch version. This difficulty is amplified because the patch itself usually does not explicitly reveal option-level dependencies such as rotation-related activation.

3.3 Key Insights of SPFuzz

To address these challenges, we present SPFuzz, a directed fuzzing framework that takes only the patch as input and jointly fuzzes both user inputs and option combinations. In the following paragraphs, we outline the key insights behind SPFuzz: how to formalize a patch into actionable fuzzing feedback, and how this feedback enables the coordinated fuzzing of options and user inputs.

Key Insight-I: *Patch behaviors provide unified guidance for directed fuzzing.* Prior studies have shown that security patches typically introduce 3 types of specific behaviors: adding sanity

checks, modifying memory-management routines, or reassigning variables—to eliminate vulnerabilities [22, 24, 33]. We further observe that these patch behaviors often implicitly expose a set of vulnerability-triggering states, i.e., variables whose runtime values directly contribute to the manifestation of a vulnerability. Based on this observation, we model patches in terms of their introduced behaviors and capture how these behaviors constrain or manipulate such variables. This abstraction enables patches with diverse fixing strategies to be handled in a unified manner and directly leveraged for PoC generation.

For example, in Figure 1, the runtime difference between `src_offset + full_bytes` and `check_buffsize` directly reflects the proximity of an input to violating the newly added sanity check. A smaller difference indicates that the input is closer to crossing the patch-enforced boundary, and therefore closer to exposing the underlying vulnerability. Building on this observation, SPFuzz systematically extracts variables manipulated by the patch and derives indicator variables that characterize vulnerability-relevant execution conditions. During fuzzing, SPFuzz monitors the runtime values of these indicator variables and uses them as guidance signals to steer execution toward vulnerability exposure.

Key Insight-II: *Patch-variable guidance enables effective joint fuzzing.* We leverage variables in the patch directly as fuzzing guidance to steer both user-input mutations and option-combination mutations toward vulnerability-exposing states. During fuzzing, mutations are prioritized if they monotonically drive state variables toward patch-imposed boundaries, such as reducing remaining size margins or increasing accumulated offsets, thereby progressively satisfying the vulnerable condition. For option combinations, we track how different options reshape the value ranges of these variables. By monitoring whether option mutations push variables toward vulnerable values, we can identify the specific options required to trigger the bug. This unified, variable-centric guidance provides a fine-grained signal that reflects patch semantics.

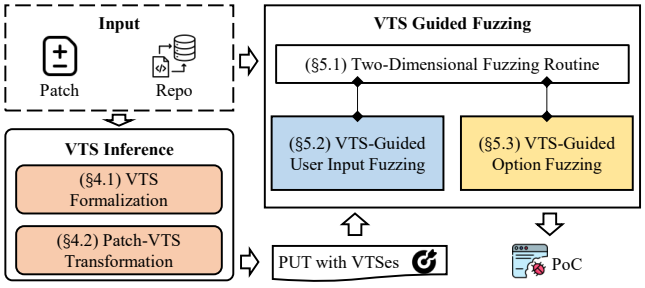


Figure 2: Overview of SPFuzz

3.4 Overview of SPFuzz

Based on the insights above, we introduce SPFuzz, a framework that integrates the ability to infer vulnerability-triggering states (VTS) from patches and jointly fuzz both options and user input to generate a full PoC, for vulnerability confirmation. As illustrated in Figure 2, SPFuzz consists of two key phases: *VTS inference* and *VTS-Guided fuzzing*.

VTS Inference. This phase consists of two primary components. First, SPFuzz formalizes the Vulnerability-Triggering State (VTS), a feedback mechanism comprising two categories of state variables designed to provide a fine-grained profile of the program states required to trigger a vulnerability. Second, SPFuzz implements a transformation-based model to systematically extract these states directly from the input patch. This process leverages a comprehensive taxonomy of patch behaviors and accounts for variable types, enabling SPFuzz to generalize across diverse patches in VTS transformation. Finally, SPFuzz instruments the program under test (PUT) with VTS mechanisms for further fuzzing.

VTS-Guided Fuzzing. In the second phase, SPFuzz employs the inferred VTS to orchestrate a coordinated, two-dimensional fuzzing process across user inputs and command-line options. By continuously monitoring the runtime values of indicator variables, the fuzzer prioritizes test cases and assigns mutation energy based on their ability to drive vulnerability-triggering. Specifically, user input mutation is steered by increases in the prime VTS, providing guidance even when control-flow signals are sparse. Concurrently, option-level exploration evaluates the effectiveness of command-line option combinations based on VTS value distributions and extrema rather than brittle distance metrics. By unifying these dimensions under a single variable-centric feedback signal, SPFuzz steers program execution toward the underlying vulnerable states.

4 Vulnerability Triggering State Inference

In this section, we define Vulnerability-Triggering States (VTS) and describe how SPFuzz infers them from patches. Our design is guided by two goals. First, VTS should provide a useful notion of progress toward a vulnerability-triggering execution in the pre-patch program. Second, the inference procedure should remain applicable across common security patch forms in a uniform solution.

4.1 VTS Formalization

Specifically, states are derived from program variables whose values directly or indirectly quantify the satisfaction degree of vulnerability-related constraints. Moreover, the state representation should be sufficiently fine-grained to sensitively capture program state changes, thereby providing informative feedback to guide fuzzing. Such feedback enables the fuzzer to prioritize user inputs and command-line option combinations based on how effectively they steer execution toward a vulnerable state.

Primary states model the satisfaction degree of vulnerability conditions originally expressed as numeric constraints. To be specific, we reformulate them into numeric states that reflect how close the condition is to being satisfied. For a given input/option combination, a larger S implies that the execution state is closer to violating the safety check in the patch, and thus closer to triggering the vulnerability. This transformation enables us to quantify the progressive satisfaction of complex conditions.

$$S^{(k)}_{prime} = \sum_{i=1}^{n_k} \alpha_i^{(k)} \cdot \mathrm{var}_i, \quad \alpha_i^{(k)} \in \{-1, +1\} \tag{1}$$

Here, k indexes one inferred primary-state component, and n_k denotes the number of numeric variables participating in that component. The coefficient $\alpha_i^{(k)}$ captures whether increasing $\texttt{var}_i$ moves the execution toward the vulnerability-triggering condition.

For example, Figure 1 presents a patch from `libtiff` that introduces a boundary check `src_offset + full_bytes >= check_bufsize` to prevent a buffer overflow. This condition is originally a Boolean constraint that determines whether the execution should abort. Following our design, we reformulate this predicate into a numeric primary state by defining S_{prime} = (`src_offset + full_bytes`)−`check_bufsize`. Under this formulation, S increases as the accumulated offset approaches the buffer limit or the memory region is shrunk. For a given input and command-line option combination, a larger S indicates that the execution state is closer to violating the safety check, and thus closer to triggering the underlying vulnerability. This numeric representation preserves the semantic intent of the patch condition while enabling progress toward the vulnerable state.

Auxiliary states are based exclusively on extreme value changes. Specifically, we only support two types of states: `MAX_CHG` and `MIN_CHG`, which capture changes in the maximum or minimum observed values of variables comprising the primary states during execution. This design choice deliberately restricts the state space to monotonic behaviors, rather than modeling arbitrary value change.

Such a design offers two benefits. First, maximum/minimum changes provide a stable and directionally consistent signal of whether program execution is moving toward a vulnerability-triggering state, which is often characterized by boundary violations or threshold crossings. Second, using `MAX_CHG` and `MIN_CHG` rather than every value avoids overfitting to transient value variations and ensures that the critical change (extreme value change following the triggering requirement) of the state can be captured during the mutation of user inputs and option combinations.

To illustrate this strategy with reference to the variables in our motivating example, specific instances of these states include `MAX_CHG(src_offset)`, `MAX_CHG(full_bytes)`, and `MIN_CHG(check_buffsize)`. In particular, `MAX_CHG(src_offset)` denotes that the maximum observed value of `src_offset` has changed during execution. For a given input/option combination, a greater number of such extremal changes indicates that the program state is progressing toward a vulnerability-triggering condition.

By combining these two types of state, our design captures both the overall satisfaction degree of vulnerability conditions and the variable-level trends that drive execution toward those conditions. This formulation provides a precise guide toward vulnerable states.

Whole Patch Integration. Building upon the state formulations for individual patch hunks, we extend our technique to the whole-patch level. We formalize the Vulnerability-Triggering State (VTS) inferred from a single patch hunk h as a tuple $VTS(h) = \langle S_{prime}^{(h)}, S_{aux}^{(h)} \rangle$. Because a vulnerability fix $\mathcal{P}$ often comprises multiple interconnected hunks, evaluating them in isolation yields incomplete guidance. Therefore, we aggregate the hunk-level states into a unified whole-patch representation, $VTS(\mathcal{P}) = \langle S_{prime}^{(\mathcal{P})}, S_{aux}^{(\mathcal{P})} \rangle$.

Specifically, to maintain semantic consistency, we align this aggregation with our fundamental state designs. The whole-patch primary state is defined as the cumulative sum of the primary states across all constituent hunks: $S_{prime}^{(\mathcal{P})} = \sum_{h \in \mathcal{P}} S_{prime}^{(h)}$. This provides a global, continuous metric of how close the execution is to satisfying all vulnerability-triggering constraints simultaneously. Conversely, the whole-patch auxiliary state is defined as the logical disjunction of all hunk-level auxiliary states: $S_{aux}^{(\mathcal{P})} = \bigvee_{h \in \mathcal{P}} S_{aux}^{(h)}$. This ensures that any critical extremal variable change across the entire patch is captured. By synergizing these aggregated states, our formulation provides the fuzzer with precise, global guidance without being trapped in the local optima of individual hunks.

4.2 Patch-VTS Transformation

Inspired by prior security patch analysis studies [22, 24, 33], we assume that security patches can generally be summarized as a set of representative behaviors. Based on this assumption, we adopt a behavior-based VTS inference approach to broadly support different types of security patches. Specifically, patches are first classified into behaviors, including adding sanity checks, variable assignments, and invoking memory-management functions.

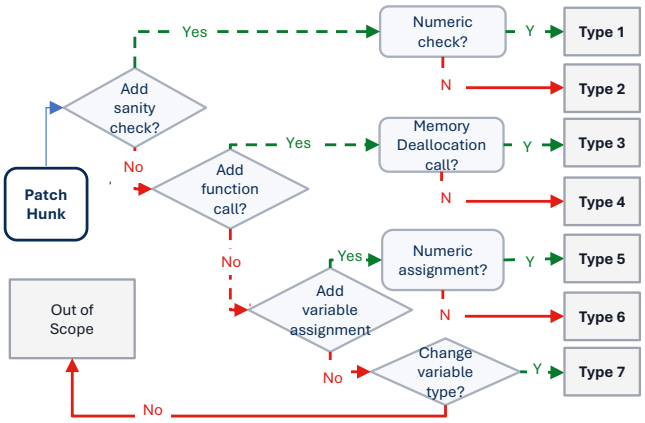


Figure 3: Patch-VTS Transformation of SPFuzz

Beyond the basic behavior, we consider the type of variables in the patch because variable types critically affect how variables can be constructed and reused during fuzzing. For example, when a patch introduces a sanity check over numeric variables, the runtime values of these variables often carry concrete semantic meaning, such as offsets, sizes, or boundary distances, that can directly indicate how close an execution is to violating a vulnerability-related constraint. In contrast, pointer variables' raw runtime values generally lack stable semantic interpretation and provide little useful guidance for fuzzing. So, we apply an additional type-aware filtering step during indicator inference. Guided by this taxonomy, SPFuzz generalizes across diverse patching patterns.

Figure 3 illustrates the structured hierarchy of our patch taxonomy. The classification process begins at the root by examining the basic syntactic changes in a code hunk and proceeds through intermediate decision nodes—such as whether a security check is being added or if the involved variables are numeric—to reach specific leaf nodes representing distinct patch types. These types serve as the foundation for our inference rules. These rules translate patch

behaviors into VTS. Based on the categories of patches, SPFuzz defines associated VTS construction strategies for each category:

- **Type 1: Adding Numeric Condition Checking.** This type covers patches that introduce or modify sanity checks over numeric variables (e.g., `if (a < b)`). For such cases, SPFuzz derives a primary state S by transforming the condition into a quantitative metric (e.g., $S = b - a$), which reflects how close an execution is to satisfying the check.
  Notably, when the check guards existing functional code in the pre-patch version rather than serving as explicit error handling, SPFuzz instead uses the additive inverse of the check condition, since the original pre-patch behavior executes only when the condition is satisfied, and the fuzzing process should therefore be guided away from satisfying the check rather than toward it.
- **Type 2: Adding Pointer Condition Checking.** These patches add checks on pointer variables, typically to prevent null-pointer dereferences. Since raw pointer values are not semantically meaningful for distances, the indicator is constructed by counting the execution frequencies of the patch point and the subsequent pointer usage.
- **Type 3: Calling Memory Deallocation.** This type includes the addition of `free()`-like function calls. SPFuzz generates a counter variable to track the execution frequency of the deallocation routine, using it as a lightweight progress signal when repeated deallocation behavior is part of the triggering context.
- **Type 4: Calling Other Functions.** This category addresses patches that introduce function calls not explicitly related to memory deallocation. To handle this, SPFuzz performs an interprocedural analysis using argument-based slicing. It maps the arguments in the pre-patch context to the parameters of the callee function and recursively analyzes the internal behavior. This recursive strategy is particularly effective for identifying encapsulated sanity checks.
- **Type 5: Assigning Numeric Variable.** When a patch modifies a numeric assignment (e.g., updating an index calculation to prevent OOB), SPFuzz monitors the magnitude of the difference between the old and new expressions. A larger difference often indicates a run more relevant to the underlying vulnerability.
- **Type 6: Assigning Pointer Variable.** This type covers pointer-state assignments (e.g., `ptr = NULL`) that alter how the pre-patch execution subsequently uses the pointer. Similar to Type 2, the inference rule focuses on the joint run frequency of the pointer assignment and its subsequent references.
- **Type 7: Changing Variable Type.** These patches modify variable data types (e.g., from `int` to `uint32_t`) to address integer overflows. The state variable is constructed by monitoring whether numeric values approach or exceed the boundaries of the original type.

Filtering Non-Security Noise. Notably, SPFuzz deliberately bypasses any patch hunk that falls outside these seven categories. Because real-world commits frequently intermingle security fixes with functionally irrelevant modifications (e.g., code refactoring or minor feature additions), this predefined taxonomy acts as a strict filter. It naturally prevents our state extraction from being polluted by the noise of non-security code changes. Importantly, this targeted approach does not compromise the tool's effectiveness: our defined categories successfully cover the vast majority of real-world vulnerability patches, a claim that will be empirically demonstrated in our evaluation.

Finally, this Patch-VTS transformation bridges the conceptual gap between static patch semantics and dynamic fuzzing guidance. By mapping categorized patch behaviors to our synergistic state representations (i.e., the combination of primary and auxiliary states), SPFuzz ensures the fuzzer receives continuous, directionally correct feedback to effectively reach the vulnerable state.

5 VTS-Guided Fuzzing

This phase details how SPFuzz integrates VTS into the main fuzzing loop. As illustrated in Algorithm 1, the fuzzing procedure jointly explores two dimensions: user inputs and command-line options. Inspired by [26], each dimension is handled by an internal loop with its own mutation. Specifically, user-input fuzzing mutates the positions and values of raw bytes, while option fuzzing mutates options with their arguments. VTS guides both dimensions by influencing input scheduling throughout the fuzzing process.

5.1 Two-Dimensional Fuzzing Routine

The two-dimensional fuzzing routine is designed to execute three primary tasks: synthesizing and maintaining coordinated input tuples, managing dimension-switching heuristics, and calculating unified VTS metrics to guide the fuzzing.

Input Tuple Initialization and Update. The process begins by initializing the seed pool S, which consists of input tuples $\langle ui, comb \rangle$ formed by independent sets of user inputs ($S.uis$) and option combinations ($S.combs$). Throughout the fuzzing campaign, SPFuzz updates these pools as new "interesting" seeds are discovered. When a mutation in either dimension yields a tuple that satisfies the seed qualification criteria—such as triggering novel code coverage or an improved VTS score—the respective components are retained in the pool. This continuous update mechanism ensures that the fuzzer maintains a high-quality corpus for both dimensions.

Dimension Switching Logic. To coordinate exploration across these dimensions, SPFuzz employs a switching heuristic based on patch reachability and execution time. The fuzzer transitions from user-input mutation to the option dimension if the target patch remains uncovered after a full cycle or if a pre-defined time limit is reached. This mechanism prevents the fuzzer from becoming trapped in sub-optimal regions of the user-input space, ensuring that it periodically explores non-default command-line configurations that may be essential for triggering the vulnerability.

For each input tuple, SPFuzz computes a composite VTS-based score that combines state value and state coverage. This score is used as a heuristic ranking signal in further fuzzing.

We first define the normalized VTS coverage as:

$$C(input) = \frac{Cover_{input}}{Cover_{max}} \tag{2}$$

Here, $Cover_{input}$ denotes the number of VTS components exercised by the current input tuple, and $Cover_{max}$ is the maximum such count observed for the current patch, which normalizes the coverage signal into the range $[0, 1]$.

Given the raw state value S_{input} observed during execution, we normalize it as:

$$\hat{S}(input) = \frac{S_{input} - S_{min}}{S_{max} - S_{min}} \tag{3}$$

The two signals are combined into a unified score:

$$Score(input) = \hat{S}(input) \cdot C(input) \tag{4}$$

This score is not a calibrated measure of exploitability; rather, it serves as a heuristic to prioritize executions that make meaningful progress toward the vulnerability-triggering conditions inferred from the patch.

5.2 VTS-Guided User Input Fuzzing

In the user input fuzzing phase, SPFuzz leverages VTS feedback to optimize both power scheduling and mutation strategies. By allocating higher energy to seeds that exhibit higher VTS and directing mutation operations toward VTS-sensitive bytes, SPFuzz efficiently steers execution toward the vulnerability-triggering state.

VTS-Guided User Input Power Assignment. At the beginning of `UserInputFuzzing` (Line 11), SPFuzz selects an input tuple $\langle ui, comb \rangle$ for mutation. Instead of relying solely on AFL-style scheduling, the power assignment is enhanced by state feedback. After an input tuple is selected, SPFuzz assigns mutation energy before performing input mutation (Line 12). The unified score is used to modulate the original AFL energy assignment:

$$power = power_{afl} \cdot (1 + Score(input)) \tag{5}$$

Besides guiding input selection with VTS feedback, SPFuzz prioritizes inputs from the user input pool with higher unified scores using an adjusted favoring index. Considering that AFL's original index has been extensively optimized for performance-related heuristics, we preserve it as the foundation of our scheduling strategy. Specifically, we integrate the VTS-derived input score into AFL's original index, ensuring that the enhanced prioritization remains performance-aware.

$$index = 0.5 \cdot index_{afl} + Score(input) \tag{6}$$

As a result, inputs exhibiting stronger indicator signals are mutated more intensively, enabling finer exploration near vulnerable state.

VTS Guided User Input Mutation. During fuzzing (Lines 13–14), SPFuzz can record the byte positions whose mutations change the auxiliary state. In subsequent fuzzing iterations, it can prioritize bit-level mutations at these recorded positions. This strategy enables the critical values to reach their maximum more efficiently, while keeping the subsequent auxiliary checks within an appropriate sensitivity range, rather than making them too sensitive.

5.3 VTS-Guided Option Fuzzing

The option fuzzing phase is dedicated to exploring the configuration dimension of the target program by identifying specific command-line parameters that influence vulnerability-triggering states. This process encompasses three core tasks: prioritizing high-utility options via a weight-based selection strategy, synthesizing semantically valid arguments through LLM-assisted mutation, and refining the option pool based on observed VTS feedback.

LLM-Assisted Option Bootstrap. Before the online option-fuzzing loop starts, SPFuzz prepares an initial pool of option combinations from project-local documentation. If a usable `man(1)` page is unavailable, SPFuzz first normalizes the binary's `-help`/`-h` output into a local manual-page-style description with a fixed prompt. It then refines candidate command templates by matching option descriptions against the patch text (commit message plus diff). These LLM-produced combinations are only bootstrap seeds: they provide high-precision starting points, while recall is achieved later through mutation and VTS-guided selection. In particular, the bootstrap prioritizes commands that first activate the target functionality family or execution mode implied by the patch, rather than merely assembling long but weakly related option strings.

Weight-Based Selection. In the `OptionFuzzing` phase (Line 19 of Algorithm 1), SPFuzz optimizes the exploration of the configuration space by prioritizing high-utility command-line parameters. For option selection (Line 20 of Algorithm 1), we adopt a weight-based strategy where each option is associated with a weight reflecting its historical effectiveness. Options that have consistently led to "interesting" behaviors, including auxiliary VTS changes, in previous iterations, are assigned higher weights, thereby increasing their selection probability in subsequent rounds. This mechanism ensures that the fuzzer concentrates its energy on the specific options most likely to influence the vulnerability-triggering state.

Mutation. SPFuzz supports a diverse set of option mutations, including insertion, removal, and argument modification. To handle the complex syntax of command-line arguments, we leverage an LLM to predict plausible value ranges and valid formats for specific options based on the program's help messages.

Qualification and Weight Updates. After mutating an option combination, SPFuzz validates its utility by replaying the set of user inputs that previously triggered auxiliary state changes (Line 23 of Algorithm 1). A mutated combination is considered "interesting" if it induces new or altered auxiliary states, at which point it replaces the original combination in the option pool and is registered into new input tuples for joint fuzzing. Finally, SPFuzz updates the option weights based on indicator feedback (Line 27 of Algorithm 1). Options that facilitate higher indicator scores or expanded VTS coverage are rewarded with larger weights, ensuring that patch-inferred vulnerability knowledge is continuously reinforced and utilized to guide fuzzing decisions.

6 Implementation

The SPFuzz fuzzing engine is built upon AFL++[11] and the CarpetFuzz-fuzzer [26] framework. Algorithm 1 summarizes the overall fuzzing procedure of SPFuzz.

Our analysis pipeline combines deterministic patch analysis with bounded LLM assistance. We adopt this hybrid design to avoid the scalability bottlenecks of traditional whole-program static analyses, such as SVF[23], which are often computationally infeasible for large-scale targets like `ffmpeg`, while still handling textual artifacts such as help messages, manual pages, and commit text. In our prototype, all LLM-assisted components in SPFuzz use the same fixed `gpt-4o-mini-2024-07-18` snapshot [2]; the implementation issues ordinary chat-completion requests with one repository-fixed prompt template per task.

Algorithm 1 Fuzzing Loop of SPFuzz

Input: Seed Tuples $S(uis, combs)$ Consists of User Inputs *.uis* and Option Combinations *.combs*,
Program Options with Weights *OPTIONS*
repeat
  S = UpdateTuples($S.uis, S.combs$)
  UserInputFuzzing(S)
  if *patchNotCovered*() **or** *VTSUnchange*() **then**
    OptionFuzzing(S, *OPTIONS*)
until *timeout* reached or *abort-signal*
Output: *CrashingInputTuple*

procedure UserInputFuzzing(S)
  $< ui, comb >$ = ChooseInputTuple(S)
  p = AssignEnergy(ui)
  for i from 1 to p **do**
    ui' = MutateInput(ui)
    if crashes **then**
      add $< ui', comb >$ to *CrashingInputTuple*
    else if isInteresting($< ui', comb >$) **then**
      add $< ui', comb >$ to S
procedure OptionFuzzing(S,*OPTIONS*)
  $comb$ = ChooseComb(S)
  opt = ChooseOption(*OPTIONS*)
  UIs' = ChooseInputSet(S)
  $comb'$ = MutateOpt($comb, opt$)
  if crashes **then**
    add $< ui, comb' >$ to *CrashingInputTuple*
  else if isInterestingComb($comb', UIs'$) **then**
    Replace($S.combs, comb, comb'$)
    OPTIONS = OptionWeightInc(*OPTIONS*, *Opt*)

LLM scope and boundary. The LLM is only given artifacts already present in the target repository or analysis workspace, including commit messages, diffs, extracted pre-patch function bodies, patch hunks, program help text/man pages, and candidate command templates. We use it for five bounded tasks: (1) normalizing CLI documentation and refining bootstrap option combinations, (2) summarizing patch-local information and classifying hunk types, (3) emitting source-level IJON-style instrumentation for the affected pre-patch functions, (4) proposing syntax-preserving repairs when the instrumented code fails to compile, and (5) leveraging the option assembler from ProphetFuzz [27] to construct option combination texts for the fuzzing engine.

For VTS instrumentation, SPFuzz leverages the LLM to generate concise, source-level annotations following the IJON-style[6] pattern, thereby exposing relevant program states to the fuzzer. We expanded the clang wrapper of AFL++ to support this custom annotation system. Importantly, the semantic analysis of patches in SPFuzz is not delegated to black-box, end-to-end model inference. The patch taxonomy, the mapping from hunk types to state-construction rules, the unified VTS score, and the online scheduling policies are all implemented by SPFuzz itself. In this architecture, the LLM acts as a bounded syntax/text analyzer and code renderer around a deterministic inference pipeline, rather than as the final decision maker of whether an execution state is vulnerability-triggering. When model outputs are noisy, the dominant failure mode is loss of useful coverage (e.g., unsupported or missed hunks, or invalid instrumentation that fails compile-time checks) rather than silent redefinition of the VTS semantics.

7 Evaluation

In this section we evaluate SPFuzz and answer the following research questions:
RQ1: How effective is SPFuzz in generating PoCs?
RQ2: How effective is SPFuzz compared with SOTA approaches?
RQ3: How much does each component of SPFuzz contribute to generate PoCs for patches?
RQ4: Can SPFuzz detect vulnerabilities by generating PoCs for real-world patches?
RQ5: How well does our patch taxonomy cover real-world security-fix commits?

7.1 Experiment Setup

Baseline. To demonstrate the effectiveness of SPFuzz, we compare it with three categories of state-of-the-art fuzzers. First, we include WAFLGo [35], a patch-oriented directed fuzzer designed for patch-based fuzzing. Second, we compare with representative directed fuzzers, including idfuzz [9], a state-of-the-art general directed fuzzer, and Lyso [7], a directed fuzzer for unknown vulnerability mining. These tools represent the closest directed-fuzzing baselines for evaluating whether target-oriented guidance can expose pre-patch vulnerabilities once the correct execution mode is available. In the directed fuzzing comparison reported later, these baselines are given the correct option configurations from public reproduction materials, so they are not disadvantaged by option discovery. Third, we compare with representative option-aware fuzzers, CarpetFuzz [26] and ZigZagFuzz [15], which explore command-line option spaces but are not designed to use patch semantics as vulnerability-triggering guidance. We report time-to-exposure (TTE), the number of exposed cases, and per-case p-values.
Dataset. To evaluate the overall effectiveness of SPFuzz on modern, real-world vulnerabilities that depend on specific command-line option combinations, we constructed a benchmark consisting of 34 CVEs along with their corresponding patches. Our dataset is derived from the 44 cases identified in § 2.2, which require non-default command-line options to trigger their PoCs. We intentionally exclude the eight default-option cases from this benchmark because they do not exercise the option-search challenge targeted by this part of the evaluation; they remain valid vulnerabilities, but including them would dilute whether a fuzzer can discover non-default command-line configurations on its own. We further refined the remaining non-default-option set to 34 cases by omitting targets that depend on GPU-specific environments, due to infrastructure constraints. Notably, all vulnerabilities in the real-world dataset were disclosed between November 2023 and April 2025. This period postdates the knowledge cutoff of the fixed `gpt-4o-mini-2024-07-18` snapshot used in this work [2], thereby mitigating potential data leakage and ensuring a rigorous evaluation on the dataset.

To evaluate the effectiveness of directed fuzzing part of SPFuzz, we use MAGMA [12], a mature and widely adopted benchmark for

evaluating fuzzing techniques. MAGMA is particularly suitable for our evaluation because its benchmark cases provide correct options and are relatively straightforward to port, reducing the risk of introducing errors during the porting process and ensuring a reliable basis for comparison with the baselines. Specifically, we select the MAGMA cases that we successfully ported to our dataset, covering vulnerabilities in TIFF, XML, and PDF processing. To further diversify the vulnerability types in our evaluation, we additionally include SSL002, a reproducible use-after-free (UAF) vulnerability from MAGMA, which we successfully reproduced in our dataset.

LLM Configuration. All LLM-assisted components in the evaluation use the same fixed `gpt-4o-mini-2024-07-18` snapshot [2] together with repository-fixed prompt templates. We do not perform per-case prompt engineering, per-target model selection, or retrieval from external vulnerability reports or PoCs. The model only receives patch-local artifacts and project-local documentation already available to the analysis pipeline.

Fuzzing Setup. Since seeds play a critical role in triggering vulnerabilities during fuzzing, and the seeds used to discover the reported cases may be manually crafted and far away from existing open-source seeds. So we use LLM-assisted initialization rather than hand-crafted reproductions. Specifically, user-input seeds are generated by the idea of G2Fuzz[36] with additional patch information, and option seeds are bootstrapped from program documentation/help and then refined using our fixed matching prompts. We treat both as bootstrap corpora rather than oracle-quality reproductions: the subsequent 24-hour campaign still relies on mutation and VTS-guided selection to reach valid PoCs, which are then distinguished from crashes through a separate validation step.

7.2 RQ1: Effectiveness

Experiment Overview. We evaluate the effectiveness of SPFuzz from two complementary perspectives using different settings. The first setting assesses the full option-aware scenario, in which a fuzzer must identify useful command-line option combinations and generate triggering user inputs. In contrast, the second setting examines the directed-fuzzing scenario under correct option configurations obtained from public reproduction materials, where the execution mode is predetermined and the remaining task is to steer user-input exploration toward the vulnerability.

Experiment Results. As shown in Table 2, in the option-aware setting, SPFuzz exposes 18 of 34 cases one more time with an average TTE of 18.24 hours. This result shows that SPFuzz can expose a substantial portion of option-dependent vulnerabilities without being given the correct command-line configuration in advance.

Overall, the results indicate that SPFuzz is effective in both evaluated settings. It can handle the harder joint search over options and inputs, and it also remains effective when the comparison is reduced to directed user-input exploration under option configurations.

Failure Analysis. Among the 16 unsuccessful cases in the option-aware setting, 5 are due to reachability/coverage issues, 10 to input-generation limitations, and 1 (CVE-2025-0840 in `objdump`) to an undocumented `-insn-width 64` argument value. The reachability/coverage cases, such as CVE-2025-1181 and CVE-2024-32228, require reaching deep execution paths, while the input-generation cases involve unusually large, highly constrained, or mutation-unfriendly inputs, e.g., CVE-2023-45322 requiring very large inputs, CVE-2025-5245 depending on a specific long-option sequence, and CVE-2023-51796 requiring a narrow internal condition.

CVE-2024-32228 in `ffmpeg` illustrates the former, as triggering the bug requires entering a deep HEVC film-grain path [3, 4]. CVE-2023-51796 illustrates the latter, as the `areverse` bug requires a specially structured audio/filter state rather than an ordinary random sample [1, 5].

Table 2: Overall results of SPFuzz and baseline option-aware fuzzers. The left block reports option-aware fuzzers. TTE denotes time-to-exposure in hours, Cnt denotes the number of successful runs out of 10, and p denotes the reported significance value for the comparison between SPFuzz and the corresponding baseline.

Program	CVE ID	SPFuzz		CarpetFuzz			ZigZagFuzz		
		TTE	Cnt	TTE	Cnt	p	TTE	Cnt	p
tiffcrop	2023-3164	<0.15	10/10	> 24.00	0/10	< 0.001	> 24.00	0/10	< 0.001
xmllint	2023-45322	> 24.00	0/10	> 24.00	0/10	1.000	> 24.00	0/10	1.000
ffmpeg	2023-49502	17.56	7/10	> 24.00	0/10	< 0.001	> 24.00	0/10	< 0.001
ffmpeg	2023-49528	17.16	5/10	> 24.00	0/10	0.007	> 24.00	0/10	0.007
ffmpeg	2023-50009	11.17	6/10	> 24.00	0/10	0.003	> 24.00	0/10	0.003
ffmpeg	2023-50010	> 24.00	0/10	> 24.00	0/10	1.000	> 24.00	0/10	1.000
ffmpeg	2023-51791	> 24.00	0/10	> 24.00	0/10	1.000	> 24.00	0/10	1.000
ffmpeg	2023-51793	17.26	7/10	> 24.00	0/10	< 0.001	> 24.00	0/10	< 0.001
ffmpeg	2023-51794	16.88	9/10	> 24.00	0/10	< 0.001	> 24.00	0/10	< 0.001
ffmpeg	2023-51795	17.44	7/10	> 24.00	0/10	< 0.001	> 24.00	0/10	< 0.001
ffmpeg	2023-51796	> 24.00	0/10	> 24.00	0/10	1.000	> 24.00	0/10	1.000
xmllint	2024-25062	> 24.00	0/10	> 24.00	0/10	1.000	> 24.00	0/10	1.000
ffmpeg	2024-32228	> 24.00	0/10	> 24.00	0/10	1.000	> 24.00	0/10	1.000
ffmpeg	2024-32229	19.88	3/10	> 24.00	0/10	0.038	> 24.00	0/10	0.038
ffmpeg	2024-32230	> 24.00	0/10	> 24.00	0/10	1.000	> 24.00	0/10	1.000
xmllint	2024-34459	15.60	5/10	> 24.00	0/10	0.007	> 24.00	0/10	0.007
objdump	2024-53589	> 24.00	0/10	> 24.00	0/10	1.000	> 24.00	0/10	1.000
xmllint	2024-56171	> 24.00	0/10	> 24.00	0/10	1.000	> 24.00	0/10	1.000
nm	2024-57360	2.67	10/10	> 24.00	0/10	< 0.001	> 24.00	0/10	< 0.001
objdump	2025-0840	> 24.00	0/10	> 24.00	0/10	1.000	> 24.00	0/10	1.000
ld	2025-1153	17.18	3/10	> 24.00	0/10	0.038	1.01	10/10	1.000
ld	2025-1176	2.71	9/10	16.02	9/10	< 0.001	0.42	10/10	1.000
ld	2025-1178	2.57	9/10	16.99	4/10	< 0.001	0.30	10/10	1.000
ld	2025-1181	13.32	7/10	20.66	5/10	0.015	0.56	10/10	1.000
ld	2025-1182	0.63	10/10	18.04	4/10	< 0.001	0.40	10/10	1.000
ffmpeg	2025-1373	18.04	5/10	19.20	2/10	0.040	> 24.00	0/10	0.007
ffmpeg	2025-1816	> 24.00	0/10	> 24.00	0/10	1.000	> 24.00	0/10	1.000
ffmpeg	2025-22921	> 24.00	0/10	19.26	2/10	0.936	> 24.00	0/10	1.000
ffmpeg	2025-25469	> 24.00	0/10	> 24.00	0/10	1.000	> 24.00	0/10	1.000
ffmpeg	2025-25471	18.74	8/10	20.76	2/10	< 0.001	> 24.00	0/10	< 0.001
ffmpeg	2025-25473	11.17	6/10	> 24.00	0/10	0.003	> 24.00	0/10	0.003
xmllint	2025-27113	> 24.00	0/10	> 24.00	0/10	1.000	13.16	5/10	0.995
objdump	2025-3198	> 24.00	0/10	> 24.00	0/10	1.000	> 24.00	0/10	1.000
objdump	2025-5245	> 24.00	0/10	> 24.00	0/10	1.000	> 24.00	0/10	1.000
Summary	-	**18.24**	**18/34**	**22.91**	**7/34**	-	**20.07**	**7/34**	-

7.3 RQ2: Comparison

Experiment Overview. We compare SPFuzz with option-aware fuzzers CarpetFuzz [26] and ZigZagFuzz [15] in the option-exploration scenario. Meanwhile, we evaluate SPFuzz against directed fuzzers WAFLGo [35], idfuzz [9], and Lyso [7] under MAGMA [12]. Together, these experiments address two complementary questions: first, whether a fuzzer can effectively generate option-dependent PoCs when the correct configuration is unknown, and second, whether it can efficiently expose the vulnerability while only the patch site is provided.

Experiment Results. The results are shown in Table 2 and Table 3. In the option-aware comparison, SPFuzz exposes 18 of 34 cases, while CarpetFuzz and ZigZagFuzz each expose 7 of 34. SPFuzz

Table 3: Comparison on the MAGMA benchmark. The 'Driver' column reports the MAGMA fuzz driver used for the corresponding target. For SPFuzz, TTE is the average time to exposure in seconds over 10 runs. For the other tools, when SPFuzz also has a numeric TTE on that bug, TTE is shown as a row-wise ratio relative to SPFuzz; when SPFuzz times out, the original numeric TTE is retained because no ratio is defined. Cnt is the number of successful runs out of 10. T.O. means that the target is not triggered within 24 hours in any run. For each bug, the best displayed TTE under this reporting rule is highlighted in bold and dark green.

Bug ID	Driver	SPFuzz		Lyso		WAFLGo		idfuzz	
		TTE	Cnt	TTE	Cnt	TTE	Cnt	TTE	Cnt
PDF003	pdfimages	**9415**	10	T.O.	0	T.O.	0	1.73x	7
PDF008	pdfimages	T.O.	0	T.O.	0	T.O.	0	T.O.	0
PDF011-1	pdfimages	**>31573**	3	T.O.	0	T.O.	0	T.O.	0
PDF016-1	pdfimages	**45**	10	113x	10	6.72x	10	3.16x	10
PDF018	pdfimages	>27606	7	T.O.	0	T.O.	0	**<0.1x**	10
PDF019-1	pdfimages	T.O.	0	T.O.	0	T.O.	0	T.O.	0
PDF021	pdfimages	**>9750**	1	T.O.	0	T.O.	0	<7.39x	1
PDF006	pdftoppm	T.O.	0	T.O.	0	T.O.	0	T.O.	0
PDF010	pdftoppm	3860	10	T.O.	0	1.48x	10	**0.9x**	10
PDF011-0	pdftoppm	**>8508**	8	T.O.	0	T.O.	0	T.O.	0
PDF016-0	pdftoppm	1218	10	7.82x	10	**0.2x**	10	0.25x	10
PDF019-0	pdftoppm	T.O.	0	T.O.	0	T.O.	0	T.O.	0
TIF002	tiffcp	T.O.	0	T.O.	0	**79856**	3	T.O.	0
TIF005	tiffcp	**800**	10	T.O.	0	98.6x	2	T.O.	0
TIF006	tiffcp	**791**	10	12.4x	10	95x	4	32.7x	2
TIF007	tiffcp	**34**	10	2.93x	10	40.8x	10	36.5x	10
TIF008	tiffcp	T.O.	0	T.O.	0	T.O.	0	T.O.	0
TIF009	tiffcp	**>11788**	6	T.O.	0	T.O.	0	<1.78x	7
TIF010	tiffcp	T.O.	0	T.O.	0	T.O.	0	T.O.	0
TIF012	tiffcp	>12964	9	<2.11x	10	<1.57x	10	**<0.38x**	10
TIF014	tiffcp	**2491**	10	5.78x	1	20.1x	9	5.99x	10
XML001	xmllint	>40493	7	**<0.64x**	10	T.O.	0	T.O.	0
XML002	xmllint	T.O.	0	T.O.	0	T.O.	0	T.O.	0
XML009	xmllint	703	10	**0.73x**	10	T.O.	0	6.68x	10
XML017	xmllint	478	10	**0.16x**	10	0.21x	10	0.29x	10
SSL002	client	**15**	10	7.93x	10	1.33x	10	2x	10
Summary	–	9030	18	15.4x	10	26.6x	11	7.14x	14

also achieves the lowest average TTE in this block: 18.24 hours, compared with 22.91 hours for CarpetFuzz and 20.07 hours for ZigZagFuzz. These results show that patch-conditioned guidance improves both exposure coverage and average exposure time compared with option-aware exploration alone.

We also report a standardized directed-fuzzing comparison on MAGMA in Table 3. Concretely, we deploy the compared fuzzers on the MAGMA-backed `poppler`, `libtiff`, and `libxml2` targets under the same 24-hour, 10-run setting, and additionally include `SSL002` as a temporal-safety case. SPFuzz's TTE is reported in seconds, and the other tools are shown as row-wise ratios relative to SPFuzz.

In this four-tool MAGMA targeting patch site comparison, SPFuzz exposes 18 of the 26 listed bugs. This exceeds idfuzz's 14 exposed bugs as well as WAFLGo's 11 and Lyso's 10. The summary row further shows that the mean case-wise TTE ratios of Lyso, WAFLGo, and idfuzz relative to SPFuzz are 15.4×, 26.6×, and 7.14×, respectively, over rows where both tools report numeric TTE. These results indicate that SPFuzz still achieves the strongest overall combination of successful run and TTE in the MAGMA experiment.

The result in Table 3 suggests that the benchmark coverage achieved by SPFuzz is still not restricted to a single bug family: the deployed `poppler`, `libtiff`, and `libxml2` targets contribute memory-boundary bugs and null dereferences, and the added `SSL002` case shows that our design can also handle a temporal-safety bug (UAF). Within the memory-boundary group, `TIF007` is also worth noting as a size/count-arithmetic-related case, since its fix changes how strip counts are derived and bounded rather than only adding a local pointer or bound check. These comparisons show that existing directed fuzzers and option-aware fuzzers each address certain scenarios of directed fuzzing, but neither category alone is sufficient for patch-based PoC generation on option-dependent vulnerabilities. By combining patch-conditioned state guidance with option-aware exploration, SPFuzz achieves stronger performance in time-to-exposure.

Failure Analysis In the case of `ld`, ZigZagFuzz outperforms SPFuzz. This is because the five cases are triggered when a single '-w' option is set, without requiring extensive user input or option combination. ZigZagFuzz performs well in this scenario as it explores more options quickly, whereas our fuzzer focuses on maintaining power in combining options and user inputs. This strategy is less effective in cases that require only short fuzzing, although it performs well in other scenarios.

At the same time, MAGMA still does not support a blanket claim that SPFuzz dominates every directed baseline on every target. Lyso is faster on several local-condition cases such as `XML001`, `XML009`, and `XML017`; idfuzz is faster on rows such as `PDF010`, `PDF016-0`, and `TIF012`; and WAFLGo is also faster on `PDF016-0` while alone exposing `TIF002`. Manual inspection of the MAGMA benchmark source suggests that part of this behavior is expected for targets where the fix site and the canary-tracked predicate are already nearly co-located. For example, in `PDF010`, the patch is a direct null check on `t3GlyphStack` immediately before the dereference, and MAGMA's canary logs precisely that null-pointer condition. In `TIF012`, the canary likewise mirrors the local condition introduced by the fix around the transfer-function metadata. `XML009` follows the same pattern: the fix guards the failed encoding switch, and MAGMA's canary logs that same local failure condition.

7.4 RQ3: Ablation Study

Experiment Overview. Given the absence of existing directed fuzzing frameworks specifically designed for command-line option exploration, our ablation study focuses on isolating the individual contributions of SPFuzz's core components. As discussed in § 4, SPFuzz enhances option exploration through two primary mechanisms: VTS-guided fitness evaluation and weight-based option selection. To evaluate their impact, we implement two ablated variants: ❶ SPFuzz $_{NoVTS}$: Disables VTS-driven feedback and relies strictly on standard code coverage to evaluate the significance of input tuples. ❷ SPFuzz $_{NoWeight}$: Removes the weight mechanism, instead selecting options for mutation with uniform probability. We re-evaluated both variants on our dataset using the same experimental parameters as the full version. All variants start from

the same user-input and option seed pools, so the performance gap isolates the effect of online guidance.

Table 4: Ablation Study Results on Our Dataset

Variant	PoCs	Efficiency Improvement
SPFuzz	18	3.0×
SPFuzz $_{NoWeight}$	11	1.8×
SPFuzz $_{NoVTS}$	6	1.0×

Experiment Results. As shown in Table 4, SPFuzz (Full), SPFuzz $_{NoWeight}$, and SPFuzz $_{NoVTS}$ successfully generated PoCs for 18, 11, and 6 patches, respectively. The full version of SPFuzz significantly outperforms both ablated variants, demonstrating that both VTS guidance and weight refinement are critical to its effectiveness. Notably, the SPFuzz $_{NoVTS}$ variant failed to generate PoCs within the given time limit for more than half of the cases that the full version successfully exposed. This underscores that in the absence of specialized guidance, the complexity of exploring a joint search space of inputs and options significantly outweighs the probability of randomly hitting a vulnerable configuration. Furthermore, these results demonstrate that even with the option combination assembled strategies employed by ProphetFuzz[27], state-of-the-art option-aware fuzzing remains insufficient for effective PoC generation. This suggests that identifying the correct configuration requires the fine-grained, state-level guidance provided by VTS, rather than static patch analysis. In other words, the gain of SPFuzz is ranking those combinations by whether they preserve progress toward the specific patch-implied vulnerable state.

The performance gap between the full version and SPFuzz $_{NoWeight}$ confirms the necessity of weight-based prioritization, which focuses mutation efforts on the subset of options most likely to influence the vulnerability-triggering state. On vulnerabilities detected by both variants, SPFuzz demonstrates a 3.1× reduction in TTE compared to SPFuzz $_{NoWeight}$, highlighting the role of weight in optimizing option exploration.

7.5 RQ4: Real-World Candidate Patch Testing

Experiment Overview. Unlike the known PoC-oriented CVE benchmark in RQ1–RQ3, this study focuses on real-world unknown cases: we select candidate security-fix commits based on patch semantics even when public PoCs are unavailable. To evaluate the real-world utility of SPFuzz, we conducted a systematic study on a curated set of such commits to determine whether actionable PoCs for the underlying vulnerabilities could be reconstructed using only the patch input and the program itself. We identified relevant commits based on the following criteria: (1) the commit message contained vulnerability-related keywords such as overflow or buffer, or mentioned automated security analysis tools like CodeQL, Coverity, indicating a memory-related security fix; (2) the patch introduced explicit logic to remediate memory-safety issues; and (3) public reproduction evidence was missing, incomplete, or only available through specialized harnesses rather than released binaries. Following these criteria, we established a target dataset comprising 110 real-world commits. Notably, in these experiments, to evaluate the ability of SPFuzz under general-purpose seed conditions, we do not use LLM-generated input seeds. Instead, we use normal seeds collected from SOTA non-directed fuzzing works [20, 21], which are derived from internet sources.

Experimental Results. Applying SPFuzz to this dataset yielded 17 distinct PoCs, demonstrating that a significant number of memory vulnerabilities remain discoverable even when their fixes are silently merged. Our findings are categorized by prior exposure in Table 6. These 17 PoCs span 11 well-tested programs, including critical utilities such as `exiv2`, `ld`, `tiffcrop`, `poppler`, and `xmllint`.

Crucially, our analysis uncovers two distinct categories of hidden security risks: cases with no prior public exposure and cases with only limited prior disclosure. Notably, the `exiv2` case in the limited-disclosure group was previously regarded as primarily observable through specialized fuzz drivers rather than production-style executions. SPFuzz successfully triggered these vulnerabilities on standard release binaries with concrete option combinations, providing stronger evidence of practical impact. These results highlight the prevalence of secretly patched vulnerabilities and demonstrate that real-world candidate security-fix commits can be turned into concrete evidence of hidden security risks in open-source software.

Table 5: RQ5 Patch-VTS transformation coverage on 549 unique security-fix commits. 'Covered' is a commit-level metric: a covered commit contains at least one hunk assigned to a concrete taxonomy type.

Project	Unique	Covered	Uncovered	textbfMissing
`binutils-gdb`	92	60	32	0
`exiv2`	17	9	8	0
`ffmpeg`	153	118	33	2
`libavif`	26	15	11	0
`libtiff`	82	59	23	0
`libxml2`	76	53	23	0
`llama.cpp`	71	52	19	0
`poppler`	32	21	11	0
Total	**549**	**387**	**160**	**2**

7.6 RQ5: Patch-VTS Transformation Comprehensiveness

Experiment Overview. To evaluate the generalizability of SPFuzz's behavior-based taxonomy in large-scale security-patch analysis, we manually curated a reliable dataset of confirmed security patches. We first compiled an audit pool of 2,396 commits from representative open-source projects (e.g., `exiv2`, `poppler`, `ffmpeg`, `binutils-gdb`, `libavif`, `libtiff`, `libxml2`, and `llama.cpp`). This pool consists of the latest 200 commits from each project alongside 796 additional commits collected via our patch-mining pipeline. Through a rigorous manual screening process, we filtered out non-security and uncertain commits, retaining 549 highly reliable security-fix commits as our ground truth. We then applied SPFuzz to this security patch set to assess its transformation comprehensiveness. Specifically, we examined the hunk-level taxonomy assignments generated by our summarizer. A commit is considered *covered* if SPFuzz successfully assigns a concrete label to at least one of its hunks, and *uncovered* if all hunks remain uncertain. This

definition is intentionally commit-level: it measures whether the taxonomy produces at least one actionable hunk label, not whether the vulnerability-triggering hunk is always identified correctly.

Experiment Results. Table 5 shows that SPFuzz's taxonomy covers 387 of 549 (70.5%) unique security-fix commits at the commit level. This should be interpreted as an upper bound on applicability at the commit level, rather than as hunk-level precision or proof that the decisive hunk is always classified correctly. Even so, the result indicates that SPFuzz can derive at least some patch-conditioned guidance from most commits in this real-world set.

To investigate the limitations, we randomly sampled 30 uncovered cases. We identified three root causes for these failures: *handling failures*, *classification failures*, and *out-of-taxonomy* behaviors. A handling failure occurs when a patch fits our taxonomy but is practically difficult to process or exploit. For example, commit `9e9d8a7` in `exiv2` fixes an OOB vulnerability by modifying a C++ `std::string` index. A classification failure happens when the LLM misses a supported pattern. For instance, commit `89f2016` introduces an explicit `(int)` cast—a clear Type 7 behavior to humans, but the LLM failed to recognize the type conversion. Finally, out-of-taxonomy cases involve completely novel repair patterns, such as commit `cbdb5e2`, which fixes an array overflow by appending a tail element using `{}`. As discussed in §4.2, only 3 of the analyzed cases fall out of scope. The vast majority are engineering limitations that can be mitigated by providing few-shot examples. The two missing cases correspond to commits that contain no code diff but only a message referencing OSS-Fuzz findings; although unusual, such commits exist in practice.

Overall, these results confirm that our behavior-based taxonomy is sufficiently expressive for the real-world patch-testing scenarios targeted by SPFuzz. While no finite taxonomy can encompass every auxiliary edit accompanying a security fix, the fact that over 70% of unique security-fix commits contain at least one successfully classified hunk suggests that our taxonomy captures a substantial portion of the dominant behavioral repair patterns essential for patch-guided vulnerability reproduction.

Possibility problem in wrong hunk-level misclassifications A wrong type assignment can change not only the quality of the signal but its nature. For example, if a value-oriented Type 1 condition is mistakenly reduced to a frequency-oriented Type 3 style signal, many seeds may start to look similarly interesting, and the campaign can degenerate toward ordinary fuzzing with much weaker directedness. We observed this effect in a manual stress test on `SSL002`: after intentionally replacing the intended VTS with a frequency-style mis-instrumentation, SPFuzz accumulated many more seemingly interesting executions and no longer maintained the same advantage over other fuzzers. This does not prove that such errors are common in our evaluated runs, but it does show that misclassification is a meaningful threat to validity and further motivates stronger validation of inferred VTS.

8 Related Work

Security Patch Identification. Identifying security-relevant patches without explicit disclosure is critical for securing the open-source software supply chain and discovering secretly patched vulnerabilities. Existing approaches generally fall into two paradigms: static program analysis, which matches code against predefined rules, and machine learning, which extracts features from commit history. However, these methods inherently suffer from false positives. Static analysis tools [19, 22, 24, 34] suffer from pattern rigidity, where benign patches are misclassified as security patches due to structural similarities. Machine learning models [29, 30, 32, 37–39] struggle with the semantic gap, often overfitting to spurious correlations (e.g., keywords or variable names) and dataset biases rather than capturing the actual vulnerability-fixing logic. Both paradigms lack the runtime verification capability to distinguish true security impacts from maintenance changes. Unlike these approaches, SPFuzz focuses on runtime validation by generating PoCs as direct evidence for pre-patch vulnerability triggering.

Directed Greybox Fuzzing. Existing directed greybox fuzzing techniques for PoC generation can be broadly categorized into three scenarios based on their guidance mechanisms. In the general directed fuzzing scenario, tools such as AFLGo [8], Beacon [13], and SelectFuzz [18] prioritize reaching target code locations by optimizing execution distances or pruning unreachable paths via static analysis; however, they often lack the semantic insight required to satisfy the condition of a vulnerability. For the specific knowledge-based scenario, methods like ChainFuzz [10], Lyso [7], CAFL [16], SDFuzz [17], and Titan [14] leverage external artifacts—such as crash sites, CVE reports, or SAST analysis results—to steer execution toward known bug patterns, but their dependency on existing seeds or detailed reports makes them inapplicable when such metadata is unavailable. Lastly, AFLChurn [40] and WAFLGo [35] target a broad set of patch-related code rather than explicitly modeling the vulnerability-triggering conditions implied by the fix. This strategy can be useful for patch-reachability testing, but in a pre-patch validation setting it may provide weaker guidance when merely reaching patch-related code is insufficient. In contrast, SPFuzz attempts to derive patch-conditioned state guidance that reflects progress toward the vulnerable behavior addressed by the patch.

Option-Aware Fuzzing. A separate line of work studies configuration and option fuzzing. Systems such as CarpetFuzz [26], ProphetFuzz [27], OSmart [28], and ZigZagFuzz [15] improve testing by mutating command-line options, inferring valid argument formats, or prioritizing configurations that increase coverage. These tools are highly relevant because they demonstrate the practical importance of option-space exploration. However, their optimization target is still general testing effectiveness rather than patch-conditioned vulnerability exposure. Consequently, they do not answer the key question in our setting: which option combinations move execution closer to the vulnerability implied by a specific security fix? SPFuzz complements this line of work by introducing a patch-derived state signal that can guide option mutations toward vulnerability-triggering behaviors instead of generic coverage.

Taken together, these three lines of work motivate SPFuzz 's design: patch-identification methods motivate the need for runtime validation, directed fuzzing provides the search backbone, and option-aware fuzzing shows why command-line configuration should be treated as a search dimension.

9 Discussion

Generalizing VTS-Guided Triggering to Two-Stage Pipelines. The utility of SPFuzz's VTS-guided triggering extends beyond standalone patch testing. Many real-world fuzzing tasks require not only reaching a deep target region, but also satisfying delicate semantic constraints after that region becomes reachable. Unlike approaches that primarily optimize global reachability, SPFuzz specializes in reconstructing the precise vulnerability-triggering state once execution enters the relevant neighborhood. Consequently, SPFuzz can be integrated as the second stage of a two-dimensional fuzzing workflow, where existing reachability-oriented fuzzers first drive execution toward the target and SPFuzz then manipulates the remaining state needed to expose the vulnerability.

LLM Role and Reliability. SPFuzz does not rely on the LLM as an end-to-end vulnerability reasoner. Instead, the model is restricted to bounded text- and syntax-processing tasks around a deterministic core, such as normalizing CLI documentation, classifying patch hunks within a fixed taxonomy, and emitting source-level instrumentation. In this design, model errors primarily reduce useful guidance coverage—for example, by missing a supported hunk type or producing instrumentation that fails validation—rather than silently changing the meaning of VTS itself. We therefore view current LLM-related limitations mainly as engineering sources of recall loss, which can be mitigated by stronger consistency checks and more robust validation, rather than as conceptual dependencies of the method.

Completeness of Patch Taxonomy. Currently, SPFuzz uses a predefined taxonomy of representative security behaviors to guide fuzzing and filter irrelevant code. This design provides precise, stable guidance across diverse patches. At the same time, our reported coverage is intentionally a commit-level measure: it indicates whether at least one hunk in a commit can be mapped to a concrete taxonomy type, not whether every relevant hunk is recognized correctly. Users can extend or adapt the taxonomy to specific environments, program domains, or vulnerability types, allowing tailored fuzzing strategies for particular scenarios without affecting the core effectiveness of SPFuzz.

About Temporal-Safety Vulnerabilities. Our current evidence suggests that SPFuzz is capable of handling at least some temporal-safety bugs, rather than being limited to spatial or numeric-boundary cases. In particular, the MAGMA results include a successful use-after-free target (`SSL002`), showing that our VTS-guided design can also help expose at least some UAF-style bugs. Looking forward, future work that targets temporal-safety bugs more explicitly could build on our framework by incorporating stronger analyses of allocation/deallocation sites, object-lifetime transitions, and use-after-free relevant pointer flows, thereby refining the current lightweight signals into more precise temporal-state guidance.

Threats from VTS Misclassification. A natural validity concern is whether the patch-to-VTS transformation might classify a hunk into the wrong behavior type and thereby distort the runtime guidance. In the current study, we did not observe confirmed cases where a patch was clearly assigned to the wrong supported type and then still counted as a successful transformation. One reason is that the taxonomy leaves many categories intentionally exclusive, and another is that our coverage policy is conservative: when the model cannot confidently match a hunk to a supported behavior, the system is more likely to leave it as uncovered than to force a speculative label. This design trades recall for precision and reduces the chance of silently introducing an obviously incorrect VTS. The risk is real as we discussed in § 4.

Overhead of State Profiling State profiling introduces nontrivial overhead, and this cost should be acknowledged explicitly. At the same time, we do not view the current cost as fundamental to the approach itself. Rather, it primarily reflects the current implementation, and we expect further reductions through tighter integration of the profiling state with AFL++'s shared-memory layout.

10 Conclusion

This paper introduces SPFuzz, a patch-oriented directed fuzzing framework designed to validate secretly patched vulnerabilities through PoC generation from candidate security patches. By formalizing patch behaviors into vulnerability-triggering states (VTS) and employing a state-guided two-dimensional fuzzing strategy, SPFuzz effectively explores the joint search space of user inputs and command-line configurations. Our evaluation on real-world datasets demonstrates that SPFuzz achieves a 51.7% success rate in PoC generation, significantly outperforming existing patch-based and general directed fuzzers. Furthermore, SPFuzz facilitated the discovery of 17 previously unknown PoCs for secretly patched vulnerabilities, four of which have been assigned CVE identifiers. This underscores SPFuzz's practical utility in turning candidate patches into concrete evidence of hidden security risks.

Acknowledgments

We thank the anonymous reviewers for the helpful comments and feedback. The work was supported in part by the National Natural Science Foundation of China (U2436207), the Shanghai Pilot Program for Basic Research — Fudan University 21TQ1400100 (21TQ012), the Guangdong Higher Education Upgrading Plan (UICR0400006-25), and the BNBU Research Grant (R0700155-26). Yuan Zhang and Min Yang are the corresponding authors. Min Yang is also with Shanghai Pudong Research Institute of Cryptology and Shanghai Institute of Intelligent Electronics & Systems.

References

[1] 2023. avfilter/f_reverse: Apply PTS compensation only when pts is available. https://github.com/FFmpeg/FFmpeg/commit/61e73851a33f0b4cb7662f8578a4695e77bd3c19.
[2] 2023. gpt-4o-mini. https://platform.openai.com/docs/models/gpt-4o-mini.
[3] 2024. avcodec/hevcdec: Fix segfault on invalid film grain metadata. https://mplayerhq.hu/pipermail/ffmpeg-devel/2024-April/325331.html.
[4] 2024. CVE-2024-32228. https://nvd.nist.gov/vuln/detail/CVE-2024-32228.
[5] 2026. FFmpeg Filters Documentation. https://ffmpeg.org/ffmpeg-filters.html.
[6] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. 2020. Ijon: Exploring Deep State Spaces via Fuzzing. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (SP)*.
[7] Andrew Bao, Wenjia Zhao, Yanhao Wang, Yueqiang Cheng, Stephen McCamant, and Pen-Chung Yew. 2025. From Alarms to Real Bugs: Multi-target Multi-step Directed Greybox Fuzzing for Static Analysis Result Verification. In *Proceedings of the 34th USENIX Security Symposium*.
[8] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*.
[9] Yiyang Chen, Chao Zhang, Long Wang, Wenyu Zhu, Changhua Luo, Nuoqi Gui, Zheyu Ma, Xingjian Zhang, and Bingkai Su. 2025. {IDFuzz}: Intelligent Directed Grey-box Fuzzing. In *Proceedings of the 34th USENIX Security Symposium*.

[10] Peng Deng, Lei Zhang, Yuchuan Meng, Zhemin Yang, Yuan Zhang, and Min Yang. 2025. CHAINFUZZ: Exploiting Upstream Vulnerabilities in Open-Source Supply Chains. In *Proceedings of the 34th USENIX Security Symposium.*
[11] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ : Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20).*
[12] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. Magma: A Ground-Truth Fuzzing Benchmark. (2020).
[13] Heqing Huang, Yiyuan Guo, Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. 2022. Beacon: Directed grey-box fuzzing with provable path pruning. In *Proceedings of the 43rd IEEE Symposium on Security and Privacy (SP).*
[14] Heqing Huang, Peisen Yao, Hung-Chun Chiu, Yiyuan Guo, and Charles Zhang. 2024. Titan: Efficient multi-target directed greybox fuzzing. In *Proceedings of the 45th IEEE Symposium on Security and Privacy (SP).*
[15] Ahcheong Lee, Youngseok Choi, Shin Hong, Yunho Kim, Kyutae Cho, and Moonzoo Kim. 2025. ZigZagFuzz: Interleaved Fuzzing of Program Options and Files. (2025).
[16] Gwangmu Lee, Woochul Shim, and Byoungyoung Lee. 2021. Constraint-guided directed greybox fuzzing. In *Proceedings of the 30th USENIX Security Symposium.*
[17] Penghui Li, Wei Meng, and Chao Zhang. 2024. {SDFuzz}: Target States Driven Directed Fuzzing. In *Proceedings of the 33rd USENIX Security Symposium.*
[18] Changhua Luo, Wei Meng, and Penghui Li. 2023. SelectFuzz: Efficient Directed Fuzzing with Selective Path Exploration. In *Proceedings of the 44th IEEE Symposium on Security and Privacy (SP).*
[19] Changhua Luo, Wei Meng, and Shuai Wang. 2024. Strengthening supply chain security with fine-grained safe patch identification. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering.*
[20] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei Han Lee, Yu Song, and Raheem Beyah. 2019. MOPT: Optimized mutation scheduling for fuzzers. In *Proceedings of the 28th USENIX Security Symposium.*
[21] Yunlong Lyu, Yuxuan Xie, Peng Chen, and Hao Chen. 2024. Prompt fuzzing for fuzz driver generation. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS).*
[22] Aravind Machiry, Nilo Redini, Eric Camellini, Christopher Kruegel, and Giovanni Vigna. 2020. Spider: Enabling fast patch propagation in related software repositories. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (SP).*
[23] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction.*
[24] Shiyu Sun, Yunlong Xing, Xinda Wang, Shu Wang, Qi Li, and Kun Sun. 2025. {DISPATCH}: Unraveling Security Patches from Entangled Code Changes. In *Proceedings of the 34th USENIX Security Symposium.*
[25] Yuan Tian, Julia Lawall, and David Lo. 2012. Identifying linux bug fixing patches. In *Proceedings of the 34th International Conference on Software Engineering (ICSE).*
[26] Dawei Wang, Ying Li, Zhiyu Zhang, and Kai Chen. 2023. {CarpetFuzz}: Automatic program option constraint extraction from documentation for fuzzing. In *Proceedings of the 32nd USENIX Security Symposium.*
[27] Dawei Wang, Geng Zhou, Li Chen, Dan Li, and Yukai Miao. 2024. Prophetfuzz: Fully automated prediction and fuzzing of high-risk option combinations with only documentation via large language model. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS).*
[28] Kelin Wang, Mengda Chen, Liang He, Purui Su, Yan Cai, Jiongyi Chen, Bin Zhang, Chao Feng, and Chaojing Tang. 2024. OSmart: Whitebox Program Option Fuzzing. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS).*
[29] Shu Wang, Xinda Wang, Kun Sun, Sushil Jajodia, Haining Wang, and Qi Li. 2023. GraphSPD: Graph-Based Security Patch Detection with Enriched Code Semantics. In *Proceedings of the 44th IEEE Symposium on Security and Privacy (SP).*
[30] Xinda Wang, Shu Wang, Pengbin Feng, Kun Sun, Sushil Jajodia, Sanae Benchaaboun, and Frank Geck. 2021. Patchrnn: A deep learning-based system for security patch identification. In *MILCOM 2021-2021 IEEE Military Communications Conference (MILCOM).*
[31] Xinda Wang, Shu Wang, Kun Sun, Archer Batcheller, and Sushil Jajodia. 2020. A machine learning approach to classify security patches into vulnerability types. In *2020 IEEE Conference on Communications and Network Security (CNS).*
[32] Bozhi Wu, Shangqing Liu, Ruitao Feng, Xiaofei Xie, Jingkai Siow, and Shang-Wei Lin. 2022. Enhancing security patch identification by capturing structures in commits. *IEEE Transactions on Dependable and Secure Computing* (2022).
[33] Hongwei Wu, Jianliang Wu, Ruoyu Wu, Ayushi Sharma, Aravind Machiry, and Antonio Bianchi. 2025. VeriBin: Adaptive Verification of Patches at the Binary Level.. In *Proceedings of the 32nd Annual Network and Distributed System Security Symposium (NDSS).*
[34] Qiushi Wu, Yang He, Stephen McCamant, and Kangjie Lu. 2020. Precisely characterizing security impact in a flood of patches via symbolic rule comparison. In *Proceedings of the 27th Annual Network and Distributed System Security Symposium (NDSS).*
[35] Yi Xiang, Xuhong Zhang, Peiyu Liu, Shouling Ji, Hong Liang, Jiacheng Xu, and Wenhai Wang. 2024. Critical code guided directed greybox fuzzing for commits. In *Proceedings of the 33rd USENIX Security Symposium.*
[36] Kunpeng Zhang, Zongjie Li, Daoyuan Wu, Shuai Wang, and Xin Xia. 2025. Low-Cost and Comprehensive Non-textual Input Fuzzing with LLM-Synthesized Input Generators. In *Proceedings of the 34th USENIX Security Symposium.*
[37] Jiayuan Zhou, Michael Pacheco, Jinfu Chen, Xing Hu, Xin Xia, David Lo, and Ahmed E Hassan. 2023. Colefunda: Explainable silent vulnerability fix identification. In *Proceedings of the 45th International Conference on Software Engineering (ICSE).*
[38] Jiayuan Zhou, Michael Pacheco, Zhiyuan Wan, Xin Xia, David Lo, Yuan Wang, and Ahmed E Hassan. 2021. Finding a needle in a haystack: Automated mining of silent vulnerability fixes. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE).*
[39] Yaqin Zhou, Jing Kai Siow, Chenyu Wang, Shangqing Liu, and Yang Liu. 2021. Spi: Automated identification of security patches via commits. In *ACM Transactions on Software Engineering and Methodology (TOSEM).*
[40] Xiaogang Zhu and Marcel Böhme. 2021. Regression greybox fuzzing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS).*

A Open Science

We open-source SPFuzz on https://github.com/spfuzz-pusher/spfuzz.

B Ethical Considerations

We responsibly disclosed discovered vulnerabilities and withheld unreleased PoCs.

C Real World Findings of SPFuzz

Table 6: Real-World PoCs Found By SPFuzz

Program	Patch (Commit ID)	Prior Exposure*
exiv2	`79ab2f6`	Limited
fax2ps	`7be20cc`	CVE-2024-13978
ld	`e2b02ac`	None
objdump	`e51fdff`	CVE-2025-8225
objdump	`656f8fb`	CVE-2025-8224
poppler	`94f62d0`	None
poppler	`998c6a7`	None
poppler	`4df3fe4`	None
tiff2rgba	`babfe37`	None
tiffcp	`87d39ea`	None
tiffcrop	`d1c0719`	Limited
tiffcrop	`9c22495`	Limited
tiffcrop	`09f53a8`	None
tiffcrop	`6357fdf`	CVE-2025-8851
tiffcrop	`82a7fbb`	Limited
xmllint	`2b0aac1`	Limited
xmllint	`37f7237`	None

* **Prior Exposure** signifies the extent of public knowledge prior to discovery by SPFuzz. *None* indicates no existing public record of the vulnerability; *Limited* indicates the issue was documented in restricted contexts (e.g., specialized fuzz drivers) but lacked an associated PoC or commit link for standard release binaries.