It Takes Two: Option-Aware Directed Greybox Fuzzing for Vulnerability PoC Generation

SUSHENG WU*, Fudan University, China
XIN HU*, Fudan University, China
YIHENG CAO, Fudan University, China
ZHUOTONG ZHOU, Fudan University, China
YIHENG HUANG, Fudan University, China
YIJIAN WU, Fudan University, China
BIHUAN CHEN†, Fudan University, China
ZHIJIA ZHAO, Fudan University, China
XIN PENG, Fudan University, China

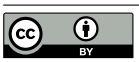
Static analysis tools can identify potential vulnerabilities, but they often fall short in providing concrete proofs-of-concept (PoCs) to validate their findings. Directed greybox fuzzing (DGF) has emerged as a promising solution by systematically guiding execution toward suspicious code locations and generating reproducible PoCs that can trigger the target vulnerabilities. However, DGF tools often overlook the influence of configurable options on reaching target locations. Besides, option-aware greybox fuzzing (GF) tools suffer from ineffective option extraction to target locations, and inefficient coordination between options and file fuzzing.

To address these limitations, we present CoupleFuzz, a novel option-aware DGF tool that redefines PoC inputs as the combination of option input (OI) and file input (FI). CoupleFuzz adopts a two-phase workflow. The static analysis phase extracts option knowledge for guiding the fuzzing. The option-aware fuzzing phase employs taint analysis to dynamically prioritize effective option combinations and file bytes to target locations, and introduces a novel *cross-guided fuzzing* strategy that coordinates OI and FI fuzzing modules and enables each module to adapt to and benefit from its counterpart's advances, iteratively driving execution toward the target locations efficiently. Our evaluation has demonstrated that CoupleFuzz significantly outperforms the state-of-the-art DGF tools in generating PoCs for 22 real-world vulnerabilities, generating 15 (a 3.1× improvement) more PoCs than the best traditional DGF baseline, with 6 0-day vulnerabilities confirmed by developers and 1 CVE identifier assigned.

CCS Concepts: • **Security and privacy → Vulnerability management**; • **Software and its engineering → Software testing and debugging**.

*Both authors contributed equally to this work.
†Bihuan Chen is the corresponding author.

Authors' Contact Information: Susheng Wu, Fudan University, Shanghai, China, College of Computer Science and Artificial Intelligence, scwu24@m.fudan.edu.cn; Xin Hu, Fudan University, Shanghai, China, College of Computer Science and Artificial Intelligence, hux24@m.fudan.edu.cn; Yiheng Cao, Fudan University, Shanghai, China, College of Computer Science and Artificial Intelligence, caoyh23@m.fudan.edu.cn; Zhuotong Zhou, Fudan University, Shanghai, China, College of Computer Science and Artificial Intelligence, zhouzt23@m.fudan.edu.cn; Yiheng Huang, Fudan University, Shanghai, China, College of Computer Science and Artificial Intelligence, yihenghuang23@m.fudan.edu.cn; Yijian Wu, Fudan University, Shanghai, China, College of Computer Science and Artificial Intelligence, wuyijian@fudan.edu.cn; Bihuan Chen, Fudan University, Shanghai, China, College of Computer Science and Artificial Intelligence, bhchen@fudan.edu.cn; Zhijia Zhao, Fudan University, Shanghai, China, College of Computer Science and Artificial Intelligence, 25213050506@m.fudan.edu.cn; Xin Peng, Fudan University, Shanghai, China, College of Computer Science and Artificial Intelligence, pengxin@fudan.edu.cn.



ACM 2994-970X/2026/7-ARTFSE098
https://doi.org/10.1145/3808105

Additional Key Words and Phrases: PoC Generation, Directed Greybox Fuzzing

ACM Reference Format:
Susheng Wu, Xin Hu, Yiheng Cao, Zhuotong Zhou, Yiheng Huang, Yijian Wu, Bihuan Chen, Zhijia Zhao, and Xin Peng. 2026. It Takes Two: Option-Aware Directed Greybox Fuzzing for Vulnerability PoC Generation. *Proc. ACM Softw. Eng.* 3, FSE, Article FSE098 (July 2026), 22 pages. https://doi.org/10.1145/3808105

1 Introduction

Static analysis tools can identify potential vulnerabilities in source code, but they often fail to provide concrete proof-of-concepts (PoCs) to validate their findings. As static analysis tools tend to generate a high number of false positives, the absence of PoCs makes vulnerability validation labor-intensive and error-prone, ultimately delaying vulnerability disclosure. Even when such vulnerabilities are acknowledged, the lack of PoC knowledge limits developers' understanding of the underlying security issues, complicates remediation, and increases the likelihood of introducing regression vulnerabilities. Directed greybox fuzzing (DGF) [1–3, 8, 9, 13–16, 19, 21, 31, 33, 40–42, 44, 46] has emerged as a powerful approach to bridge this gap by systematically steering execution toward suspicious target locations and generating reproducible file input that can trigger the target vulnerability.

Yet, DGF faces a fundamental but under-explored challenge, i.e., the critical role of configurable options in reaching target locations. Real-world software systems often expose numerous configurable options (e.g., command-line flags such as "-a -l -R" in *tiffcrop* [28]), which can significantly influence program semantics. Such options often determine whether the execution path leading to a vulnerability is even reachable, and thus directly affect the effectiveness and efficiency of DGF. For example, for libtiff-740 [25], static analysis tools correctly identified the vulnerable code, but DGF under default options (commonly no option provided) failed to trigger this vulnerability because the vulnerable execution path was entirely disabled by default. This illustrates a key insight that *specific options are often a necessary precondition for reaching the target location*. For another example of CVE-2022-43249 [6], when we applied DAFL [16], a state-of-the-art DGF tool that does not account for options, it required 1.39×10^5 mutations to first reach and crash the vulnerable location under default options. The reason is that the default options redirected the vulnerable function pointer to a safe one, detouring execution away from the vulnerability. However, once the "-0" option was enabled, this restriction was lifted, and the vulnerability was triggered after only 7.9×10^4 mutations. This demonstrates a deeper insight that *the efficiency of DGF can be heavily influenced by options*.

While configurable options have been studied in coverage-based greybox fuzzing (GF) [17, 18, 36–38, 43], no prior work has investigated option awareness in the context of DGF. Existing option-aware GF tools fall into two categories, document-based tools [17, 18, 36, 37, 43], which rely on developer documentation, and code-based tools [38], which extract options directly from source code. Both face fundamental challenges when applied to DGF.

Challenge 1: Ineffective Option Extraction to Target Locations. Current option-aware GF tools lack effective reasoning about which options are relevant for reaching target locations. In detail, document-based tools [17, 36, 37, 43] extract options from documentation, which may be incomplete and irrelevant for target locations. Though code-based tools [38] extract options directly from source code to mitigate option incompleteness, they can still include many irrelevant options that cannot influence the semantic toward target locations. As a result, these tools often waste substantial computational resources selecting and mutating ineffective options that have no impact on triggering the target vulnerability. Furthermore, if two or more options are selected, these existing tools do not consider the effectiveness of relations between those different options to target locations, leading to inefficient exploration of the option combination.

Challenge 2: Inefficient Coordination between Option and File Fuzzing. Existing option-aware GF tools struggle to effectively coordinate option fuzzing and file fuzzing, leading to inefficient

exploration. The limitations manifest in two folds. First, some tools (e.g., [43]) mutate options and files simultaneously. While this strategy improves path coverage, it obscures the source of contribution; i.e., whether the progress is driven by option mutation or by file mutation. Without this attribution, the fuzzer cannot provide meaningful feedback to guide subsequent scheduling. Second, later tools (e.g., [18, 36–38]) separate option and file mutation, which allows them to identify the source of coverage gains more precisely. However, they fail to propagate this knowledge across domains; i.e., option-level feedback is not leveraged to guide file mutation, and file-level feedback is not used to refine option selection and mutation. This one-sided reasoning leaves exploration shallow, often trapping GF tools in local optima and missing deeper execution paths to target locations that require coordinated interactions between option mutation and file mutation.

Goal. Our goal is to achieve effective and efficient option-aware DGF that can reliably generate reproducible *PoC inputs* for potential vulnerabilities. Here, *potential vulnerabilities* refer to suspicious code locations identified through static analysis or manual security audits, where neither the vulnerability's exploitability nor the exploit options are known a priori. Compared with reproducing known vulnerabilities whose exploit options are already available, this setting poses a fundamentally harder problem, as the fuzzer must simultaneously discover effective *option input* (*OI*) and *file input* (*FI*) combinations. Unlike prior DGF tools that treat the *PoC input* solely as FI (i.e., the program input as a file), we redefine it as the combination of OI and FI, since vulnerabilities often remain unreproducible without proper options. This broader definition captures the practical conditions needed for vulnerability validation, and enables DGF to generate PoC inputs that closely resemble real-world conditions.

Approach. To address these challenges, we present a novel fuzzer, named CoupleFuzz. CoupleFuzz adopts a two-phase workflow: (1) a *Static Analysis* Phase that systematically examines how options impact program semantics, and (2) an *Option-Aware Fuzzing* Phase that tightly coordinates OI fuzzing and FI fuzzing. The first phase aims to extracts option knowledge for later option effectiveness inference. To achieve this, CoupleFuzz maps each option to its directly-impact variables (DIVs), thereby grounding configurable options in concrete program variables. CoupleFuzz further analyzes the complete value space of each DIV along with the specific control conditions under which each value is assigned. We define an option as *effective* if its DIVs can influence the data or control flow that leads to the target location. However, because multiple options can modify the same DIVs, the effectiveness of a given option may be overwritten by other options, hindering its intended impact on the flow to the target location. To this end, CoupleFuzz extracts holistic option constraints by analyzing how options may overwrite each other's DIVs.

The second phase alternately fuzzes the OI and FI part of a seed to clearly attribute how each part contributes to reaching the target location. During each OI fuzzing round, CoupleFuzz performs dynamic taint analysis to track the runtime usage of all DIVs. It then maps these actively used DIVs back to their corresponding options, allowing CoupleFuzz to precisely identify which options are truly effective in influencing the current execution path. Based on the effective options, CoupleFuzz refine and update the option constraints previously extracted during static analysis. As a result, CoupleFuzz infers which combinations of options are truly effective for reaching the target location, thereby tackling **Challenge 1**. Unlike prior approaches that treat OI and FI fuzzing independently, CoupleFuzz employs a novel *cross-guided fuzzing* strategy that coordinates OI fuzzing and FI fuzzing via mutual feedback. The core idea is to leverage execution insights from one input part to guide mutations of the other. Specifically, CoupleFuzz records execution paths before and after each OI (or FI) mutation, then analyzes the path differences to identify whether an interesting execution path is discovered based on its distance to the target location. Unlike traditional DGF, which assigns energy only to FI when discovering interesting execution paths, CoupleFuzz first determines whether the deeper execution path is relevant to the counterpart and can benefit from cross-input guidance by

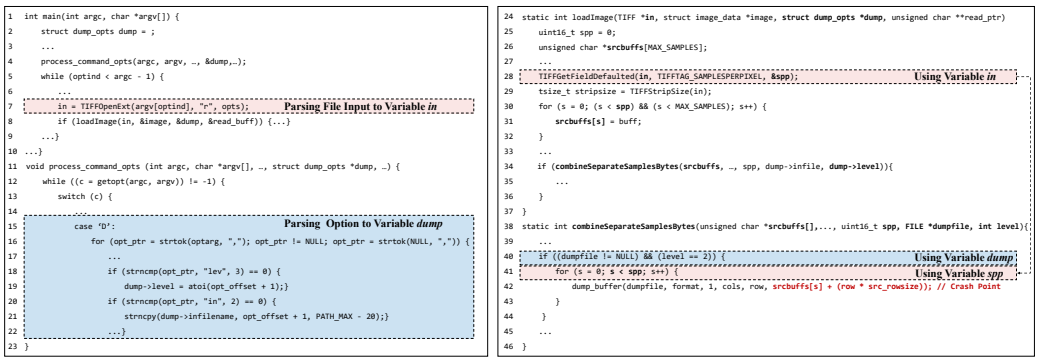


Fig. 1. A Buffer Overflow Vulnerability (i.e., libtiff-740) Disclosed in *libtiff*

Table 1. PoC Generation Time for the Two Motivating Vulnerabilities

Vulnerability	DGF			Option-Aware GF		Option-Aware DGF
	DAFL	WindRanger	AFLGo	ProphetFuzz	ZigZagFuzz	CoupleFuzz
libtiff-740	T.O	T.O	T.O	9.6h	20.5h	**2.1h**
libde265-345	C.E	2.5h	6.1h	T.O	T.O	**0.3h**

applying taint analysis at each part. When relevance is confirmed, CoupleFuzz efficiently allocates energy to mutate the relevant portion of the counterpart to explore the deeper execution path more efficiently. This *cross-guided fuzzing* strategy enables FI (OI) of a seed to adapt to and benefit from the advances of its counterpart, creating a synergistic feedback loop that systematically drives executions toward the target location, thereby tackling **Challenge 2**.

Evaluation. We conducted comprehensive experiments to evaluate the efficiency of CoupleFuzz across two scenarios, option-unavailable and option-available datasets. For the option-unavailable scenario, we curated a dataset of 26 vulnerabilities from static analysis reports. CoupleFuzz significantly outperformed existing baselines, generating 15 more PoCs (a 3.1× improvement) than the best traditional DGF baseline (DAFL [16]). Compared to the best option-aware GF baseline (ProphetFuzz [37]), CoupleFuzz also generated 15 more PoCs (a 3.1× improvement) with an average speedup of 1.3×. Notably, CoupleFuzz achieved the best performance in 15 out of 26 evaluated cases; and in 9 cases where all other tools failed to generate any PoCs, only CoupleFuzz succeeded. We reported 13 0-day vulnerabilities along with the generated PoCs to the developers, resulting in 6 confirmations and 1 CVE identifier assignment. For the option-available scenario, we collected a dataset of 40 vulnerabilities from the literature. CoupleFuzz remained highly competitive, achieving the best performance in 18 out of 40 cases and generating 9 PoCs that no other tool could produce. Our ablation study showed the contribution of both FI fuzzing and OI fuzzing modules, with disabling either module leading to a significant performance degradation or complete failures in generating PoCs for 21 and 19 cases, respectively.

Contributions. This paper makes the following contributions.

- We propose a novel option-aware DGF approach that extracts option knowledge and coordinates option input (OI) and file input (FI) fuzzing, enabling efficient PoC generation.
- We conduct comprehensive evaluation to demonstrate CoupleFuzz's superior efficiency in both option-unavailable and option-available PoC generation scenarios.

2 Motivation

We present two real-world vulnerabilities that illustrate (1) why option awareness is essential for DGF to generate PoC input (hereafter, simply PoC for brief) and (2) the unique challenges that current option-aware GF tools encounter in the DGF scenario to generate PoC.

Traditional DGF Tools Fail or Become Inefficient without Proper Options. Our first example is a buffer overflow vulnerability in *libtiff* [25], shown in Fig. 1. This vulnerability, initially detected by CodeQL, arises from an out-of-bounds array access at Line 42 when the variable `spp` exceeds the allocated size of `srcbuffs`. CoupleFuzz successfully generated a PoC that triggered this vulnerability, which was subsequently confirmed and patched by the developers of *libtiff*. We initially ran state-of-the-art DGF tools (i.e., DAFL [16], WindRanger [9], and AFLGo [2]), using Line 42 as the target location. As shown in Table 1, none of these tools succeeded in generating a PoC within 24 hours. Our investigation revealed that their file input (FI) seeds consistently reached the conditional checks at Line 40, but could not bypass them, because the default option configuration omitted the crucial "-D" option, which made `dumpfile` and `level` a default value of `NULL` and `0` respectively. Only when "-D" was enabled with arguments "level:lev input:in" to initialize `dumpfile` and set `level` to 2, the execution path could then reach Line 42. *This demonstrates that options can be a necessary precondition for reaching the target location.*

Our second example is CVE-2022-43249 (a.k.a. libde265-345) in *libde265* [6], illustrated in Fig. 2. It arises because the function `put_epel_hv_fallback` is initially assigned to the function pointers `accel->put_hevc_epel_h_8` (Line 40) and `accel->put_hevc_epel_h_16` (Line 42). If no option is configured, the condition `l >= 20` is satisfied by the default value of `l` of 10,000 (Line 36), the function `init_acceleration_functions_sse` is called, and the assignment to `accel->put_hevc_epel_h_8` is overwritten with the safe function pointer (Line 45). After this overwrite, only the function pointer `accel->put_hevc_epel_h_16` remains vulnerable and can be further triggered. The complete call chain is ①→②→③→④→⑤. Although WindRanger and AFLGo can generate an option-free PoC within 24 hours (see Table 1), their efficiency is still lower than that of CoupleFuzz. The reason lies in how option configuration influences path exploration. When the "-0" option is provided (highlighted in blue box at Line 6, 12, and 13), the variable `l` is instead set to `0` (Line 13). This prevents execution from reaching Line 45, thus the function pointer `accel->put_hevc_epel_h_8` is never overwritten (⑥→⑦→④). As a result, the vulnerable pointer remains active and can be invoked at Line 20. In fact, WindRanger generated a seed that can reach Line 20 within the first 40 minutes, but because the option was not configured, the invocation at Line 20 could not trigger the vulnerability. However, if the same seed is executed with the "-0" option, the vulnerability is immediately triggered, while without it, the fuzzer must continue searching for a more complex path involving `accel->put_hevc_epel_h_16`. *This demonstrates that although options are sometimes not strictly necessary for target reachability, they can significantly accelerate PoC generation.*

Challenge 1: Ineffective Option Extraction to Target Locations. We then investigate whether current option-aware GF tools can generate the correct PoC. Consider the first example in Fig. 1, even the most effective tool ProphetFuzz [37] required nearly 9.6 hours to generate the PoC, as reported in Table 1, demonstrating its capability but lacking efficiency compared with CoupleFuzz. After analyzing the seeds generated by ProphetFuzz, we observed that over 95% of the 31,831 generated seeds did not contain the effective option input (OI), specifically the argument "level:lev input:in". Similar inefficiencies occurred with other option-aware GF tools. This is because *libtiff* supports 38 configurable options, yielding 2^{38} possible combinations, most of which are irrelevant to this specific vulnerability. Although ProphetFuzz employed a large language model to extract potentially

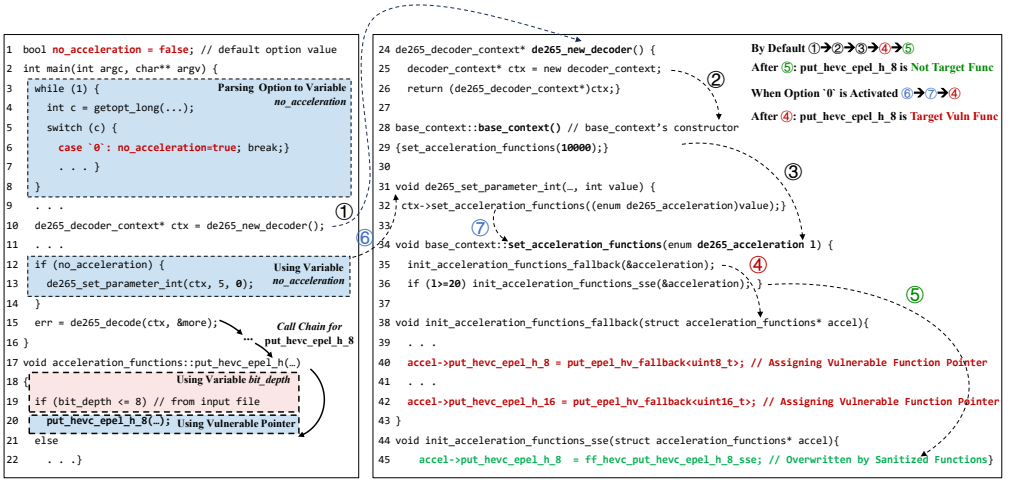


Fig. 2. A Buffer Overflow Vulnerability Disclosed in *libde265*

dangerous option combinations, the majority of selected combinations failed to include the essential "-D" option required for triggering this vulnerability. These observations highlight **Challenge 1**.

Challenge 2: Inefficient Coordination between Option and File Mutation. We next examine whether current option-aware GF tools can efficiently generate the correct PoC when both option input (OI) and file input (FI) must be considered. Our analysis reveals a critical coordination problem that severely impacts fuzzing efficiency. In Fig. 1 and 2, we highlight the FI impact in red boxes and the OI impact in blue boxes. For the first example in Fig. 1, FI is derived into the variable `in` (Line 7), which is then used to extract the number of samples per pixel (`spp`) via `TIFFGetFieldDefaulted` (Line 28). This `spp` value later determines the memory allocation for `srcbuffs` at the target location. ZigZagFuzz employs a strategy of mutating FI and OI alternately and independently. It successfully bypasses the conditional check at Line 40 by mutating OI, allowing execution to reach the `for` loop at Line 41. However, the `spp` value remains too small to trigger an overflow. In the subsequent FI mutation turn, ZigZagFuzz shifts focus to other irrelevant bytes, failing to affect `spp` and thus not reaching the overflow. This occurs because the tool only considers coarse-grained coverage feedback and lacks fine-grained guidance on which FI bytes influence `spp`. Consequently, the progress gained from OI mutation is wasted. A mirror problem arises in the second example in Fig. 2. At Line 19, the `bit_depth` value is determined by FI (the derivation process is omitted for brevity). Here, ZigZagFuzz successfully bypasses this conditional check by mutating FI, reaching Line 20. However, the function pointer does not point to the vulnerable target function. In the next OI mutation turn, the tool fails to recognize that the function pointer is influenced by the option "-0," and thus the vulnerability is not triggered. Conversely, the progress from FI mutation is wasted.

The unawareness of the interplay between FI and OI is common across all option-aware GF tools, which becomes highly inefficient in the DGF scenario. A more efficient approach would be to leverage fine-grained feedback from each counterpart. For the first example, OI mutation reaches Line 41, which should then guide FI mutation toward bytes influencing `spp` to trigger the overflow. For the second example, FI mutation reaches Line 20, which should then guide OI mutation to adjust the option that directs the function pointer to the vulnerable target. The central difficulty, however, lies in inferring whether a variable like `spp` originates from FI, OI, neither, or both, which is a very challenging task. These observations reveal a fundamental coordination problem; i.e., current tools cannot leverage feedback from OI (FI) mutations to guide FI (OI) mutations. This lack of cross-guided coordination substantially degrades fuzzing efficiency, highlighting **Challenge 2**.

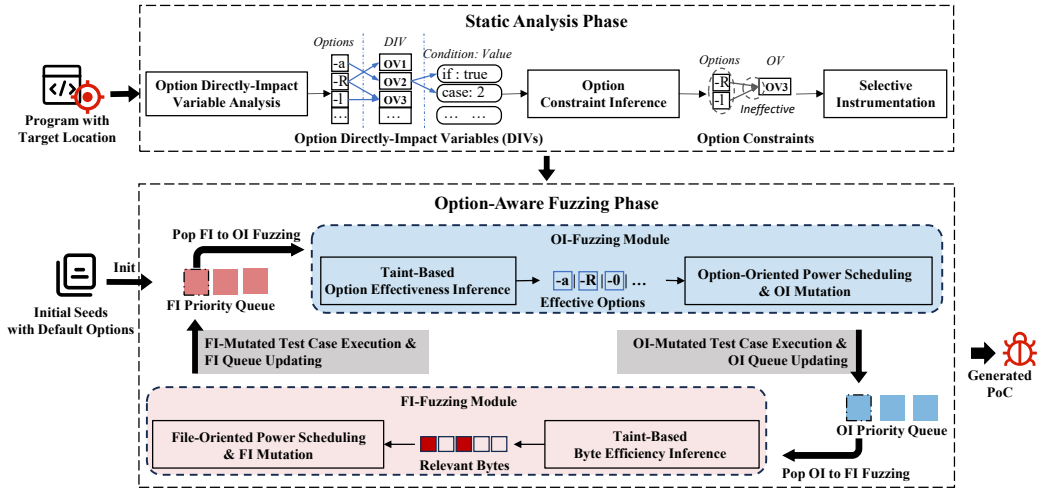


Fig. 3. Approach Overview of CoupleFuzz

3 Approach

As shown in Fig. 3, CoupleFuzz adopts a two-phase workflow: (1) a *Static Analysis* Phase that examines how options impact program semantics, and (2) an *Option-Aware Fuzzing* Phase that coordinates option input (OI) and file input (FI) fuzzing. An overview of these two phases is introduced in Sec. 1.

3.1 Static Analysis Phase

3.1.1 Option Directly-Impact Variable Analysis. Given the target location in a program, CoupleFuzz begins by extracting each option, identifying its directly-impact variables (DIVs), and collecting all possible values assigned to these DIVs from the option parsing routine in the program.

Identifying Options and DIVs. CoupleFuzz leverages the state-of-the-art option-aware GF tool Osmart [38] to extract all options $\mathcal{O}$ and their DIVs $\mathcal{D}$. Here, $\mathcal{O}$ and $\mathcal{D}$ are many-to-many mapping; i.e., a single option may influence multiple DIVs, and a DIV may be affected by multiple options.

Example 3.1. Fig. 4 shows the option parsing function in *tiffcrop*. The extracted options $\mathcal{O}$ = {`-f`, `-l`, `-s`, `-t`} and DIVs $\mathcal{D}$ = {`deffillorder`, `outtiled`, `deftilelength`}. In fact, there are 38 options and 71 DIVs in *tiffcrop* respectively, and the others are omitted for brevity.

Value Space Analysis of DIVs. CoupleFuzz analyzes how each option influences the values of its associated DIVs. CoupleFuzz first parses the program's intermediate representation (IR) to identify all assignment instructions (e.g., `store`, `memcpy`, `memset`) that set values for each DIV $div \in \mathcal{D}$ and collects its assigned values. Since these assignments are often governed by control dependencies, CoupleFuzz performs backward control-dependency analysis (using SVF [34]) from each assignment instruction of div, and extracts the predicates of controlling branches (e.g., `br` conditions, `switch` case) and conjoins them into a branch condition. CoupleFuzz then pairs each possible value v with its corresponding branch condition bc, forming a comprehensive value set $V_{div} = \{(v, bc)\}$, where v is a possible value and bc is the branch condition under which v is assigned.

Based on this DIV value space analysis, CoupleFuzz establishes a mapping from each option to its associated DIVs, capturing both the complete value space and the precise enabling conditions. This mapping is formally defined as $\mathcal{M}(opt) = \{(div, V_{div}) \mid div \in \mathcal{D}$ and div is opt's DIVs$\}$, where $opt \in \mathcal{O}$. By making both the value space and its control conditions explicit, we provide direct and actionable guidance for targeted option mutation during fuzzing (see Sec. 3.2.2).

Example 3.2. For the example in Fig. 4, CoupleFuzz extracts the option `-f` and DIV `deffillorder`, and parses the program IR to locate the assignment operations that assign values to `deffillorder`

```
1 void process_command_opts{
2     while ((c = getopt(argc, argv)) != -1) {
3         switch (c) {
4           ……
5           case ‘f’:
6              if (streq(optarg, "lsb2msb"))
7                  *deffillorder = FILLORDER_LSB2MSB;
8              else if (streq(optarg, "msb2lsb"))
9                  *deffillorder = FILLORDER_MSB2LSB;
10          case ‘l’:
11             outtiled = TRUE;
12            *deftilelength = atoi(optarg);
13          case 's’:
14             outtiled = FALSE;
15          case 't':
16             outtiled = TRUE;
17          case ...: } } }
```

Fig. 4. Option Parsing Function in *tiffcrop*

Algorithm 1: Option-File Coordinated Directed Fuzzing

Input: Instrumented program $\mathcal{P}_{\mathcal{I}}$, Initial seeds $\mathcal{S}$, Target location $\mathcal{T}$, Option mapping $\mathcal{M}$, Option constraints $\mathcal{C}$

```
1  Queue_FI ← INIT(S);
2  Queue_OI ← empty;
3  Crashes ← ∅;
4  while not timeout do
     // OI-Fuzzing Phase:
5    s ← Queue_FI.POP(); // s=<OI, FI>
6    IOpts ← TAINT_ANALYSIS(s);
7    EOpts ← OI_INFERENCE(s.OI, IOpts, C);
8    foreach eopt ∈ EOpts do
9      e ← ASSIGN_ENERGY_OI(s, eopt);
10     for i = 1 to e do
11       OI' ← MUTATE_OI(eopt, M(eopt));
12       EXECUTE(⟨OI', FI⟩);
13       if OI' has gain then
14         Queue_OI.INSERT(OI', FI);
15       if OI' crashes at T then
16         Crashes.ADD(OI', FI);
17   Queue_OI.INSERT(s);
     // FI-Fuzzing Phase:
18   s ← Queue_OI.POP(); // s=<OI, FI>
19   EBytes ← TAINT_ANALYSIS(s);
20   e_eff, e_rest ← ASSIGN_ENERGY_FI(EBytes, s.FI);
21   for i = 1 to e_eff do
22     FI' ← MUTATE_EBYTES(EBytes);
23     EXECUTE(⟨OI, FI'⟩);
24     ... // Same Subsequent Operations in OI-Fuzzing
25   for i = 1 to e_rest do
26     FI'' ← MUTATE_FI(s.FI);
27     EXECUTE(⟨OI, FI''⟩);
28     ... // Same Subsequent Operations in OI-Fuzzing
29   Queue_FI.INSERT(s);
```

at Line 7 and 9. The macros `FILLORDER_LSB2MSB` and `FILLORDER_MSB2LSB` are expanded to the constants 2 and 1, respectively. The control conditions for these assignments are `optarg = lsb2msb` at Line 6 and `optarg = msb2lsb` at Line 8. Hence, $\mathcal{M}$(`-f`) is denoted as $\{(deffillorder, \{(2, \texttt{optarg} = \texttt{lsb2msb}), (1, \texttt{optarg} = \texttt{msb2lsb})\})\}$.

3.1.2 Option Constraint Inference. An option is considered *effective* if its associated DIVs impact the data or control flow toward the target location. However, multiple options may influence the same DIVs, and when options are enabled simultaneously, later DIV assignments can overwrite earlier ones, rendering the former options ineffective. For instance, as illustrated in Fig. 4, the DIV `outtiled` can be set by either the `-s` or `-t` option. After option parsing, the final value of `outtiled` depends on which option is specified last. If both options are provided (e.g., `-s -t`), the value assigned by the latter option (`-t`) overwrites the former, rendering the earlier assignment ineffective. To address this challenge, CoupleFuzz performs holistic analysis of DIV overwriting. This ensures that only option combinations with non-overwritten influence on program behavior are considered.

Holistic DIV Constraint Extraction. For each option $opt_i \in O$, CoupleFuzz extracts all its DIVs as $DIV(opt_i) = \{\, m.div \mid m \in \mathcal{M}(opt_i) \,\}$. It then considers all subsets of O, i.e., $\mathcal{P}(O)$. For any option set $OPT = \{opt_1, opt_2, \ldots, opt_n\} \in \mathcal{P}(O)$, the combined DIV set is defined as $DIV(OPT) = DIV(opt_1) \cup DIV(opt_2) \cup \cdots \cup DIV(opt_n)$. Two option sets conflict when their combined DIV sets are identical, because this indicates overwritten effects caused by DIV overwriting. Formally, the DIV constraint for an option subset OPT_j is defined by Eq. 1.

$$C(OPT_j) = \Big\{ OPT_k \;\Big|\; \forall OPT_k \in \mathcal{P}(O) \setminus \{\emptyset, O\},\ DIV(OPT_j) = DIV(OPT_k),\ OPT_j \not\subseteq OPT_k,\ OPT_k \not\subseteq OPT_j \Big\} \quad (1)$$

We collect all possible option constraints for each option subset in $\mathcal{P}(O)$, denoted as C.

Example 3.3. For the example in Fig. 4, CoupleFuzz observes that both `-s` and `-t` assign values to the same DIV (`outtiled`). As a result, enabling both options together leads to the latter option

will overwrite the former. Thus, $C(\{\texttt{-s}\}) = \{\{\texttt{-t}\}\}$. It is important to note that although `-l` also sets `outtiled`, it is not considered as conflicting with either `-s` or `-t` in this context, because `-l` additionally sets another DIV, `deftilelength`. The combination of `outtiled` and `deftilelength` can result in program behaviors to the target location that are distinct from those produced by modifying `outtiled` alone. Therefore, OPT_j whose DIV sets merely overlap or are subsets of OPT_k are not considered as conflicting; and only those with identical DIV sets are treated as conflicting.

3.1.3 Selective Instrumentation. CoupleFuzz further optimizes the fuzzing process by selectively instrumenting only the basic blocks that are reachable to the target location following the approach of SDFuzz [21] to provide focused coverage feedback.

3.2 Option-Aware Fuzzing Phase

The workflow of CoupleFuzz's fuzzing phase is presented in Algorithm 1. At a high level, CoupleFuzz takes as input an instrumented program $\mathcal{P}_I$, a set of initial seeds $\mathcal{S}$ with default options (e.g., empty or `-input`) to ensure the program are executable, a target location $\mathcal{T}$, a map of options and their DIVs $\mathcal{M}$, and a set of option constraints C from static analysis, and returns a set of crashing test cases found. These crashing test cases are validated as the PoCs for the target vulnerability following the prior methodology [16]. Different from traditional DGF tools, CoupleFuzz considers a seed s as a pair of option input OI and file input FI, i.e., $s = \langle OI, FI \rangle$.

CoupleFuzz first initializes both seed queues and the set of crashing test cases (Line 1-3). Given the initial seeds $\mathcal{S}$, CoupleFuzz explores as many potential paths as possible to reach the target location, following the approach of AFLGo [2]. The explored file inputs are stored in the seed queue $Queue_{FI}$, prioritized by their distance to the target location (Line 1). Additionally, CoupleFuzz performs a preliminary exploration by testing each file input with default options to capture any early crashes that occur in shallow execution paths. The seed queue $Queue_{OI}$ is initialized as empty (Line 2). CoupleFuzz then enters the main fuzzing loop (Line 4-29). $Queue_{FI}$ and $Queue_{OI}$ serve as prioritized queue for file and option input mutations, respectively, enabling CoupleFuzz to alternate and coordinate exploitation between file structure and option space for more efficient fuzzing.

OI-Fuzzing Module. The main idea of this module is to fuzz efficient OIs that can leverage the interesting execution paths from FI part of the current seed to reach deeper or more diverse paths to the target location. CoupleFuzz begins by selecting a seed from $Queue_{FI}$ (Line 5). To understand which options can influence the current seed's execution semantics, CoupleFuzz performs DIV-sensitive taint analysis on the seed (Line 6). If at least one of the option's DIVs is used in the execution path, this option is considered to be able to *impact* the current program semantics, and all such options are denoted as $IOpts$. However, as discussed in Sec. 3.1.2, some of these impactful options may overwrite the DIV values set by the current OI, thereby overwriting the effect achieved by the existing OI. To address this concern, CoupleFuzz refines the option constraints C to identify effective options $EOpts$ (Line 7). These effective options are capable of influencing the execution semantics of the current seed without compromising the beneficial effect of the existing OI. Next, CoupleFuzz allocates energy to each option $eopt \in EOpts$ by the relevance to the interesting execution paths gained from FI fuzzing (Line 9) to decide mutation attempts. CoupleFuzz then mutates $eopt$ and executes the resulting mutated seed with new OI' (Line 11-12). If the mutated seed reaches deeper or more diverse paths to the target location, it is inserted to $Queue_{OI}$ (Line 14). Notice that the insertion does not rewind the pointer to the next seed to be selected. Thus this algorithm is free from starvation. If it crashes at the target location $\mathcal{T}$, it is added to $Crashes$ (Line 16). Finally, the original seed is reinserted into $Queue_{OI}$ (Line 17), ensuring seeds without efficient OI discoveries can be reconsidered in subsequent iterations in FI-Fuzzing module.

FI-Fuzzing Module. The main idea of this module is to fuzz efficient bytes in FI that can leverage the interesting execution paths from OI part of the current seed to reach deeper or more diverse paths to the target location. CoupleFuzz begins by popping a seed from $Queue_{OI}$ (Line 18), and then performs taint analysis on the seed to extract the set of impactful FI bytes *EBytes* which are relevant to the interesting execution paths from OI part of the current seed (Line 19). Next, CoupleFuzz balances the energy between *EBytes* and the holistic $s.FI$ (Line 20). For *EBytes* and $s.FI$, CoupleFuzz applies different mutation strategies (Line 22 and 26), executes the resulting mutated seed, and updates $Queue_{FI}$ and saves the crashes like the OI-Fuzzing Module (Line 24 and 28). Finally, the original seed is reinserted into $Queue_{FI}$ (Line 35).

3.2.1 Taint-Based Option Efficiveness Inference. Taint analysis guides fuzzers towards efficient mutation and help explore hard-to-reach branches [11]. However, traditional solutions are labor-intensive and slow. CoupleFuzz adopts a lightweight DIV-sensitive taint analysis to extract effective options.

Taint-Based Impactful Option Extraction. When a DIV is used along the execution path, this indicates that the DIV has a direct impact on program behavior. In such cases, CoupleFuzz marks all options associated with that DIV as impactful. Specifically, CoupleFuzz performs inter-procedural backward slicing to trace each DIV $div \in \mathcal{D}$ back to its earliest definition, rather than simply tainting DIVs at their assignment in the option parsing function. At each origin definition point, CoupleFuzz assigns a unique taint flag $flag$ to div. During execution, CoupleFuzz monitors the usage of div, and propagates its taint flag through all variables influenced along the execution path, continuing this propagation until program termination. Formally, for each div, CoupleFuzz collects all variables tainted by div along the execution path, resulting in the DIV-to-variable taint map $\mathcal{M}_{taint}(div) = \{v_1, v_2, \ldots, v_n\}$, where each variable v_j is directly or transitively influenced by div during execution.

CoupleFuzz uses the option-to-DIV map $\mathcal{M}$ and the DIV-to-variable taint map $\mathcal{M}_{taint}$ to efficiently determine which options can affect program behavior in the current execution. For each option opt_j, if $\mathcal{M}_{taint}(m.div)$ is empty for $\forall m \in \mathcal{M}(opt_j)$, then opt_j has no effect on the current execution, and can be skipped in further mutations; otherwise, opt_j is considered as impactful. CoupleFuzz collects all such impactful options into *IOpts* for targeted mutation. Since options that were previously effective may become ineffective after FI part mutation, CoupleFuzz leverages *IOpts* to dynamically shrink the option space for $s.OI$, ensuring that only currently impactful options are considered for further mutation.

Effective Option Extraction. During OI mutation, CoupleFuzz adopts a focused strategy of adding only one new option to the current option input $s.OI$, while keeping all previously selected options unchanged (see Sec. 3.2.3 for further discussion of this strategy). However, as previously noted, not every newly added option will necessarily have an effect to the target location and some may be ineffective, rendering the mutation unproductive. To ensure that only meaningful mutations are performed, CoupleFuzz refines the option constraints C to filter *IOpts* and produce a subset of truly effective options, denoted as *EOpts*. Only these effective options are considered for further mutation, thereby improving the efficiency and precision of the fuzzing process.

For each impactful option $opt_j \in IOpts$, let its impactful DIVs be $DIV_j = \{m.div \mid m \in \mathcal{M}(opt_j)\}$. CoupleFuzz considers only options containing these impactful DIVs to construct refined constraints and yields C^+ following the same constraint extraction process as C (see Sec. 3.1.2).

To ensure that adding an option $iopt \in IOpts$ does not break the effectiveness of existing options in the current seed $s.OI$, CoupleFuzz validates that $iopt$ does not conflict with any subset of options in the current seed. Formally, let *SOpt* denote the set of currently enabled options in the seed. An option $iopt \in IOpts$ is considered effective if Eq. 2 is satisfied,

$$\forall S \subseteq SOpt: \quad C^+(\{iopt\} \cup S) \cap SOpt = \emptyset \tag{2}$$

where S is any subset of the currently enabled options $SOpt$. CoupleFuzz collects all such effective options into $EOpts$. Each $eopt \in EOpts$ can independently contribute to program behavior, thereby reducing unnecessary computation and focusing mutation efforts on truly impactful options.

Example 3.4. Following Example 3.2, the option `-s` initially appears effective when combined with the option `-l`, since `-l` can affect the unique DIV `deftilelength`. However, after performing taint analysis, CoupleFuzz finds that `deftilelength` is not used in the current execution path. Thus, the refined constraints $C^+(\texttt{-s}) = \{\texttt{-l}, \texttt{-t}\}$ are generated, and `-l` is excluded if `-s` is already enabled.

3.2.2 Option-Oriented Power Scheduling. Traditional DGF tools typically focus all resources on mutating the FI part of a seed, which, as shown in Sec. 1, is often ineffective and inefficient. To address this, CoupleFuzz introduces an energy allocation strategy that dedicates a portion of fuzzing effort to the OI part. However, the optimal division of energy between OI and FI remains an open question: allocating too little to OI yields minimal benefit, while allocating too much undermines FI exploration. CoupleFuzz determines energy allocation to OI based on the efficiency of options in $EOpts$. For those discovered new execution paths that have closer distances to the target location mutated by FI, options that influence DIVs used in these paths are considered efficient and receive higher energy allocation. This strategy is motivated by the observation that these newly discovered paths represent promising avenues toward the target, and targeted OI scheduling can further exploit these paths to drive execution even closer to the target, achieving faster convergence.

CoupleFuzz first examines which newly explored program behaviors are influenced by options. Specifically, it compares the inter-procedural control flow graph (ICFG) paths before and after FI mutation: $\Delta\xi(s) = \xi(s) - \xi(s')$, where $\xi(s)$ is the execution path of current seed and $\xi(s')$ is the path before FI mutation. We define two sets for each option $opt_i \in EOpts$ by Eq. 3 and 4.

$$EB(opt_i) = \{bb \mid bb \in \Delta\xi(s),\ \exists m \in \mathcal{M}(opt_i) :\ \exists var \in \mathcal{M}_{taint}(m.div) :\ var \in bb\} \tag{3}$$

$$B(opt_i) = \{bb \mid bb \in \xi(s),\ \exists m \in \mathcal{M}(opt_i) :\ \exists var \in \mathcal{M}_{taint}(m.div) :\ var \in bb\} \tag{4}$$

Here, bb is the basic block in the execution path, $EB(opt_i)$ is the set of newly covered basic blocks after FI mutation that are tainted by opt_i, and $B(opt_i)$ is the set of all basic blocks in the current execution path $\xi(s)$ tainted by opt_i.

CoupleFuzz adopts the simulated annealing algorithm from AFLGo for option-oriented power scheduling, with a modified energy calculation based on the new seed distance metric. We define the average distance from tainted basic blocks to the target basic block T_b by Eq. 5,

$$d_{opt}(opt_i, T_b) = \frac{\sum_{bb \in EB(opt_i)} d_b(bb, T_b)}{|EB(opt_i)|} \ \text{ if } EB(opt_i) \neq \emptyset, \ \text{ else } \ \frac{\sum_{bb \in B(opt_i)} d_b(bb, T_b)}{|B(opt_i)|} \tag{5}$$

where $d_b(bb, T_b)$ denotes the distance from the basic block bb to the basic block of the target location T_b. The metric d_{opt} prioritizes options that taint basic blocks closer to the target, especially focusing on newly explored blocks $EB(opt_i)$ if available; otherwise, it considers all tainted blocks $B(opt_i)$. A smaller d_{opt} value indicates that mutating opt_i is more likely to help reach the target, and thus, more fuzzing energy should be allocated to it.

Example 3.5. As illustrated in Fig. 2, CoupleFuzz first taints the definition of `no_acceleration` at Line 1. After one round of FI mutation, CoupleFuzz discovers newly covered basic blocks, including Line 12. CoupleFuzz then identifies that option "-0" (where $\mathcal{M}(\texttt{-0}) = \{\texttt{no\_acceleration}\}$) can influence the current program behavior. CoupleFuzz then calculates the distance from the tainted basic blocks to the target location T_b and allocates energy (e.g., 1,223) to option "-0".

3.2.3 Option-Oriented OI Mutation. CoupleFuzz considers one strategy for OI mutation, i.e., adding only one new option to the current option input $s.OI$, while keeping all previously selected options unchanged. This is because: (1) the current OI has already demonstrated effectiveness for the seed in the previous OI mutation round, so changing it could disrupt its beneficial impact; and (2) mutating multiple options simultaneously introduces combinatorial complexity, making it hard to attribute improvements to specific options. By adding one option at a time, CoupleFuzz can clearly assess each option's individual contribution and maintain a focused fuzzing process. Specifically, given an option $opt \in EOpts$, CoupleFuzz performs structural mutations on the OI part of the seed. Each OI is represented as a tuple $\langle eopt, arg_1, arg_2, \ldots \rangle$, where $eopt$ denotes the effective option and $arg_1, arg_2, \ldots$ are its associated arguments. After adding opt, CoupleFuzz mutates its arguments, following the approach of ZigZagFuzz [18], enabling the generation of diverse and valid OIs.

During OI mutation, CoupleFuzz exploits the value space of each option's DIV to optimize energy allocation. For each $eopt \in EOpts$, CoupleFuzz examines all associated DIVs by iterating over $m \in \mathcal{M}(eopt)$, where the value space for each DIV div is given by $m.V_{div} = \{(v, bc)\}$, with v as a possible value and bc as its enabling condition. CoupleFuzz systematically enumerates all v for div, collecting the set $V = \{v \mid (v, bc) \in m.V_{div}\}$. If $|V|$ (the number of possible values) is less than the assigned energy budget, CoupleFuzz terminates mutation early after all values are explored, reclaiming unused energy for subsequent fuzzing. To further avoid wasteful exploration, CoupleFuzz analyzes the reachability of each value by checking its control condition bc. If any control condition bc is determined by configurable arguments (such as `optarg` or `argv[1]`) and is parsed by functions like `atoi` or `strtol`, CoupleFuzz proactively mutates the relevant arguments simultaneously. This ensures that the mutated arguments can satisfy the feasible control condition bc, thereby maximizing the likelihood of exercising meaningful value combinations.

Example 3.6. Following Example 3.5, consider the option `no_acceleration` whose value space is limited to $\{true\}$ which is much smaller than the assigned energy budget of 1,223. After one round of OI mutation, CoupleFuzz terminates the mutation process and the unused energy is efficiently reclaimed and redirected to subsequent fuzzing.

3.2.4 Taint-Based Byte Efficiency Inference. Building on the OI power scheduling strategy, CoupleFuzz applies FI-aware taint analysis to assess the efficiency of individual bytes in the FI. Specifically, it identifies bytes that influence any variable along those discovered new execution paths that have closer distances to the target location mutated by OI, as indicated by $\Delta\xi(s)$ before and after OI mutation. These bytes, considered *efficient*, are collected into the set $EBytes$ for targeted mutation.

3.2.5 FI-Oriented Power Scheduling. Different from traditional DGF tools that treat all bytes equally, CoupleFuzz allocates more energy to these efficient bytes for more focused exploration. Formally, CoupleFuzz computes the average distance from all basic blocks in the current execution paths $\xi(s)$ and the previous path before OI mutation $\xi(s')$, respectively to the target location T_b by Eq. 6,

$$d_{FI}(s, T_b) = \frac{\sum_{bb \in \xi(s)} d_b(bb, T_b)}{|\xi(s)|}, \quad d_{FI}(s', T_b) = \frac{\sum_{bb \in \xi(s')} d_b(bb, T_b)}{|\xi(s')|} \tag{6}$$

where s is the current seed and s' is the seed before OI mutation. Based on these distance measurements, CoupleFuzz computes an efficiency score as $score_{eff} = d_{FI}(s', T_b) - d_{FI}(s, T_b)$ to assess whether the newly discovered basic blocks are closer to the target than the previous ones. When $score_{eff} > 0$, the current execution path $\xi(s)$ is closer to the target than the previous path $\xi(s')$, indicating that OI mutation has successfully advanced the fuzzing process toward the target location.

To determine the appropriate energy allocation for efficient bytes $EBytes$ and to adequately explore the newly discovered basic blocks gained from OI mutation, CoupleFuzz normalizes this

efficiency score to obtain a weight for *EBytes* by Eq. 7,

$$w_{eff} = \sqrt{\max\left(0, \frac{|EBytes|}{|Bytes|} \cdot \frac{d_{FI}(s', T_b) - d_{FI}(s, T_b)}{d_{FI}(s', T_b)}\right)} \tag{7}$$

where $|Bytes|$ represents the total number of file input bytes. The ratio $\frac{|EBytes|}{|Bytes|}$ quantifies the efficiency of byte exploration: when $|Bytes|$ is small, the input can be completely explored with minimal energy, while larger $|Bytes|$ requires proportionally more energy to achieve comprehensive exploration. The distance improvement factor, $\frac{d_{FI}(s', T_b) - d_{FI}(s, T_b)}{d_{FI}(s', T_b)}$, measures how much closer these new basic blocks bring execution to the target, prioritizing mutations that yield real progress. The square root in w_{eff} smooths the combined effect of these factors, preventing excessive energy allocation when both are too low. If $score_{eff}$ is not positive, w_{eff} is set to zero, and no extra energy is spent on these bytes. The total energy e is then split: $e_{eff} = w_{eff} \cdot e$ targets efficient bytes *EBytes*, while $e_{rest} = e - e_{eff}$ is used for general FI mutation following AFLGo [2]. This targeted energy allocation strategy ensures that fuzzing effort is concentrated on the most promising bytes, maximizing the likelihood of advancing toward the target location.

3.2.6 FI-Oriented Mutation. CoupleFuzz employs a fine-grained mutation strategy for *EBytes* (i.e., iteratively flipping individual bits or performing small arithmetic modifications). This strategy maintains the structural integrity of the seed, preserving the progress achieved through OI mutation, while enabling targeted exploration for further incremental advances toward the target.

Example 3.7. Following Example 3.5, after one round of OI mutation, CoupleFuzz enables "-0" during OI fuzzing. In FI fuzzing, CoupleFuzz identifies the bytes related to `bit_depth` and applies a fine-grained mutation strategy to them, aiming to adjust `bit_depth` to lower than 8.

4 Evaluation

We implemented the static analysis module of CoupleFuzz by SVF [34], and implemented its option-aware fuzzing module by extending AFL v2.57b [30] and DFSan [7], totally adding approximately 5,500 lines of C/C++ code and 1,000 lines of Rust code. We designed the following research questions to evaluate the performance of CoupleFuzz. All experiments were performed on a machine with an Intel(R) Xeon(R) Silver 4314 CPU, and 64 GB of memory, running Ubuntu 22.04. Each fuzzer was executed within an isolated Docker container to ensure a consistent and controlled environment.

- **RQ1 Efficiency Evaluation on Option-Unavailable Dataset.** How is the efficiency of CoupleFuzz in PoC generation for option-unavailable dataset?
- **RQ2 Efficiency Evaluation on Option-Available Dataset.** How is the efficiency of CoupleFuzz in PoC generation for option-available (commonly provided by human experts) dataset?
- **RQ3 Ablation Study.** How are the contributions of the FI-Fuzzing module and the OI-Fuzzing module to the achieved efficiency of CoupleFuzz?

Baselines. We compared CoupleFuzz with three state-of-the-art DGF tools, i.e., DAFL [16], WindRanger [9], and AFLGo [2]. We excluded Hawkeye [3] and ParmeSan [32] since they are close sourced. Besides, we also compared CoupleFuzz with two state-of-the-art option-aware GF tools, i.e., ProphetFuzz [37] and ZigZagFuzz [18]. We excluded POWER [17] and AFL++-argv [10] from our comparison, as ZigZagFuzz is a direct enhancement of both tools, and its original evaluation has already demonstrated consistent improvements over them. We also excluded CarpetFuzz [36] as ProphetFuzz is a direct enhancement of it. Furthermore, despite our attempts to contact the developers, the fuzzing module of OSmart [38] remained unavailable, rendering it unusable.

Evaluation Metric. Following previous work [9], we adopted *time to exposure* (TTE) as our primary evaluation metric. TTE measures the time required for a fuzzer to generate the first input

that triggers a crash at a specific target location only when the time limit was not reached. This metric provides a direct assessment of the efficiency of each fuzzer in generating PoCs for the vulnerabilities. Additionally, we introduce the *success ratio* (Succ.) metric, which measures the ratio of successful PoC generations out of all trials.

Initial Seeds and Time Budget. The initial seeds can affect the efficiency of the fuzzing process. We utilized the seeds provided by the project itself and those available across different fuzzers to ensure a fair comparison following DAFL [16]. Each tool was allocated a time budget of 24 hours. To account for the inherent randomness of fuzzing and reduce the impact of fluctuations, we repeated the fuzzing process 10 times for each tool and reported the average TTE across these runs.

4.1 Efficiency Evaluation on Option-Unavailable Dataset (RQ1)

Dataset Construction. To assess the performance of CoupleFuzz on option-unavailable dataset, we curated a dataset of 26 vulnerabilities with their target locations from static analysis reports through a rigorous three-step process.

Step 1: Static Vulnerability Report Generation. To ensure a comprehensive evaluation, we selected a diverse set of 15 real-world programs that are widely recognized as benchmarks in prior fuzzing research [2, 9, 16]. For each of the 15 programs, we collected multiple versions released after January 2020, resulting in a total of 220 program versions. These programs were deliberately chosen to encompass a wide range of functionalities and codebase sizes, thereby allowing us to thoroughly assess the efficiency of CoupleFuzz. For each program version, we applied CodeQL [4] with a set of carefully curated rules [12] to generate static vulnerability reports. We totally collected 979 static vulnerability reports with dangerous target locations for subsequent analysis.

Step 2: False Positive Filtering. Given the relatively high false positive rate of CodeQL [16, 23, 35, 45], running DGF on all static vulnerability reports would be prohibitively expensive. To address this, we first randomly sampled 276 static vulnerability reports (with a confidence level of 95% and a margin of error of 5%) and filtered out duplicates for cross-version vulnerabilities, yielding 69 reports. Specifically, when the same vulnerability (i.e., sharing identical target location with same vulnerability type) is reported by CodeQL in multiple program versions, we retain only the newest affected version to reduce redundant fuzzing effort. Two security experts, each with over five years of experience, then independently examined these reports to identify and remove false positives. Disagreements were resolved by a third expert to ensure consensus. As a result of this process, we obtained a final set of 37 unique and confirmed static vulnerability reports, requiring approximately 100 hours of effort and achieving a Cohen's Kappa coefficient of 0.927.

Step 3: PoC Generation. We conducted fuzzing experiments using the three DGF tools and two option-aware GF tools from our baselines, as well as CoupleFuzz. For the DGF tools, we specified the dangerous target locations identified by CodeQL as fuzzing targets, and supplied the default options to ensure the programs could execute correctly (e.g., `-i –input file, -o –output file`). Following DAFL [16], we only considered PoCs whose crash types and crash lines are identical to those described in the static vulnerability reports. Through this rigorous and systematic process, we successfully generated PoCs for 26 vulnerabilities, ensuring that for each vulnerability, at least one PoC was generated by at least one tool and successfully triggered the corresponding vulnerability.

Overall Results. As shown in Table 2, CoupleFuzz generates 22 (59%) PoCs, outperforming the best baseline, DAFL (7), by 3.1×. Among these 22 PoCs, CoupleFuzz achieves the *best success ratio* in 21 cases, which is 5.3× more than the best baseline DAFL (4). For the 6 vulnerabilities where both CoupleFuzz and DAFL successfully generated PoCs, CoupleFuzz achieves an average PoC generation time of 7.3 hours, which is 0.84× of DAFL's speed due to the overhead of option exploration. CoupleFuzz still outperforms other baselines, achieving speedups of 1.2× to 1.6× over Windranger, AFLGo, ProphetFuzz, and ZigZagFuzz. These results demonstrate that the slight

Table 2. PoC Generation Results on Option-Unavailable Dataset (T.O denotes the fuzzer was unable to generate PoC within the time budget, C.E denotes compilation error, and O.E denotes the fuzzer was unable to extract the options. #Best denotes the number of cases for which the tool has the best performance among all other tools. O-DGF denotes the option-aware DGF tool, DGF denotes the traditional DGF tool, O-GF denotes the option-aware GF tool. **w/o FI.** denotes the TTE of CoupleFuzz ablated with FI-Fuzzing module. **w/o OI.** denotes the TTE of CoupleFuzz ablated with OI-Fuzzing module. **w/o Sele.** denotes the TTE of CoupleFuzz ablated with selective instrumentation. TTE is calculated only when the time limit (24h) was not reached, and averaged across successful runs. Succ. denotes the number of successful PoC generations out of 10 trials.)

Program	ID	DGF						O-GF				O-DGF							
		DAFL		Windranger		AFLGo		ProphetFuzz		ZigZagFuzz		CoupleFuzz		w/o FI.		w/o OI.		w/o Sele.	
		Succ.	TTE	Succ.	TTE	Succ.	TTE	Succ.	TTE	Succ.	TTE	Succ.	TTE	Succ.	TTE	Succ.	TTE	Succ.	TTE
tiffcrop	1	0/10	T.O	0/10	T.O	0/10	T.O	**1/10**	**23.9**	0/10	T.O	0/10	T.O	0/10	T.O	0/10	T.O	0/10	T.O
	2	0/10	T.O	0/10	T.O	0/10	T.O	**10/10**	1.3	0/10	T.O	**10/10**	**0.6**	0/10	T.O	0/10	T.O	10/10	0.6
	3	0/10	T.O	0/10	T.O	0/10	T.O	7/10	9.6	4/10	20.5	**9/10**	**2.1**	0/10	T.O	0/10	T.O	7/10	2.8
	4	0/10	T.O	0/10	T.O	0/10	T.O	0/10	T.O	0/10	T.O	**5/10**	**14.1**	0/10	T.O	0/10	T.O	5/10	16.7
	5	0/10	T.O	0/10	T.O	0/10	T.O	0/10	T.O	0/10	T.O	**10/10**	**0.9**	0/10	T.O	0/10	T.O	10/10	1.1
	6	0/10	T.O	0/10	T.O	0/10	T.O	2/10	17.2	0/10	T.O	**6/10**	**13.6**	0/10	T.O	0/10	T.O	4/10	15.9
	7	0/10	T.O	0/10	T.O	0/10	T.O	0/10	T.O	0/10	T.O	**5/10**	**16.0**	0/10	T.O	0/10	T.O	5/10	18.4
	8	0/10	T.O	0/10	T.O	0/10	T.O	0/10	T.O	0/10	T.O	**6/10**	**11.5**	0/10	T.O	0/10	T.O	3/10	12.5
mp4encrypt	1	0/10	T.O	0/10	T.O	0/10	T.O	O.E	O.E	0/10	T.O	**3/10**	**19.1**	0/10	T.O	0/10	T.O	0/10	T.O
	2	0/10	T.O	0/10	T.O	0/10	T.O	O.E	O.E	4/10	13.6	**6/10**	**8.6**	0/10	T.O	0/10	T.O	5/10	10.2
img2sixel	1	C.E	C.E	0/10	T.O	0/10	T.O	0/10	T.O	0/10	T.O	**1/10**	**19.6**	0/10	T.O	0/10	T.O	0/10	T.O
	2	C.E	C.E	0/10	T.O	0/10	T.O	0/10	T.O	**10/10**	**1.0**	**10/10**	2.2	0/10	T.O	0/10	T.O	10/10	1.0
	3	C.E	C.E	0/10	T.O	0/10	T.O	**1/10**	23.8	**1/10**	22.8	**1/10**	**21.7**	0/10	T.O	0/10	T.O	0/10	T.O
	4	C.E	C.E	4/10	**8.6**	**5/10**	16.4	0/10	T.O	0/10	T.O	0/10	T.O	0/10	T.O	0/10	T.O	0/10	T.O
	5	C.E	C.E	0/10	T.O	0/10	T.O	0/10	T.O	0/10	T.O	**5/10**	**14.2**	0/10	T.O	0/10	T.O	4/10	14.2
	6	C.E	C.E	0/10	T.O	0/10	T.O	**2/10**	**23.6**	0/10	T.O	0/10	T.O	0/10	T.O	0/10	T.O	0/10	T.O
	7	C.E	C.E	0/10	T.O	0/10	T.O	0/10	T.O	0/10	T.O	**5/10**	**13.3**	0/10	T.O	0/10	T.O	5/10	13.6
	8	C.E	C.E	**10/10**	**0.1**	**10/10**	0.3	0/10	T.O	**10/10**	0.4	**10/10**	0.2	0/10	T.O	10/10	0.1	10/10	0.39
dec265	1	C.E	C.E	0/10	T.O	0/10	T.O	0/10	T.O	0/10	T.O	**10/10**	**0.3**	10/10	0.1	0/10	T.O	10/10	0.3
bsdcpio	1	7/10	**4.1**	6/10	4.7	7/10	8.8	0/10	T.O	0/10	T.O	**10/10**	5.2	0/10	T.O	0/10	T.O	10/10	5.6
	2	8/10	5.5	9/10	7.4	6/10	**5.0**	0/10	T.O	0/10	T.O	**10/10**	6.0	0/10	T.O	0/10	T.O	10/10	6.6
tiff2ps	1	**10/10**	**0.2**	**10/10**	0.4	**10/10**	0.5	0/10	T.O	0/10	T.O	**10/10**	0.5	0/10	T.O	10/10	0.4	10/10	0.6
tiff2pdf	1	3/10	16.7	0/10	T.O	0/10	T.O	1/10	19.7	0/10	T.O	**6/10**	**16.6**	0/10	T.O	7/10	21.1	7/10	18.4
tiffdither	1	**4/10**	**13.6**	0/10	T.O	0/10	T.O	0/10	T.O	0/10	T.O	0/10	T.O	0/10	T.O	0/10	T.O	0/10	T.O
rizin	1	**6/10**	**8.8**	3/10	18.8	1/10	21.7	0/10	T.O	0/10	T.O	2/10	12.7	0/10	T.O	3/10	11	2/10	14.3
jasper	1	**10/10**	**1.1**	**10/10**	1.6	**10/10**	1.9	O.E	O.E	0/10	T.O	**10/10**	2.5	0/10	T.O	10/10	2.1	10/10	2.9
#Best		4	5	3	2	4	1	4	2	3	1	**21**	**15**	–	–	–	–	–	–

time overhead from option exploration is a worthwhile trade-off; i.e., CoupleFuzz generates 3.1× more PoCs and achieves 5.3× more *best success ratio*, clearly showcasing CoupleFuzz's superior effectiveness in PoC generation compared to the best state-of-the-art.

Comparison with Traditional DGF. For Windranger and AFLGo, CoupleFuzz generates 2.7× more PoCs than each baseline. CoupleFuzz also achieves a speedup by 1.2× and 1.4×, respectively. CoupleFuzz achieves the best performance in 15 cases, outperforming the best DAFL (5) by 3×.

Comparison with Option-Aware GF. Compared to ProphetFuzz, the best option-aware GF baseline, CoupleFuzz generates 3.1× more PoCs and achieves an average speedup of 1.3×. Regarding ZigZagFuzz, CoupleFuzz outperforms it by generating 4.4× more PoCs and achieving an average speedup of 1.6×. Furthermore, CoupleFuzz achieves the best performance in 15 cases, outperforming the best option-aware GF baseline, ProphetFuzz (2), by 7.5×.

In-Depth Analysis. To better evaluate the performance of CoupleFuzz, we categorize all generated PoCs of CoupleFuzz based on their triggering requirements: (1) PoCs generated for *Option-Dominated* vulnerabilities, which require specific program options and can be triggered without any file input, including only 1 case in our dataset; (2) PoCs generated for *File-Dominated* vulnerabilities, which can be triggered using default options, including 6 cases; (3) PoCs generated for *Hybrid* vulnerabilities, which require both specific program options and crafted file inputs, including 15 cases; and (4) PoCs that could not be generated by CoupleFuzz, including 4 cases.

PoCs for Option-Dominated Vulnerabilities (1). In this category, dec265-1 [24] is a vulnerability that can only be triggered by specifying the `-T` option, which is not included in the program's default options. CoupleFuzz efficiently discovers this required option through its OI-fuzzing phase and generates a PoC in just 0.3 hours, whereas all traditional DGF tools fail because they rely solely on default options and cannot reach the target location. This underscores the necessity of option-awareness in DGF for generating PoCs for such vulnerabilities.

PoCs for File-Dominated Vulnerabilities (6). In this category, CoupleFuzz can achieve competitive or even surpasse the efficiency of the best traditional DGF tool, DAFL. For instance, consider the case of tif2pdf-1 [26], a NULL pointer dereference vulnerability that can be triggered solely by providing a specially crafted TIFF file. CoupleFuzz is able to generate a PoC in 16.6 hours, which is slightly faster than DAFL (16.7 hours). This efficiency can be attributed to CoupleFuzz's ability to automatically identify and leverage effective option combinations (e.g., `-j` which ensures that the vulnerable function is exercised in this case) during the OI-fuzzing phase. Combining with the efficient FI-fuzzing phase, CoupleFuzz is able to rapidly generate the necessary input file, thereby significantly streamlining the entire fuzzing process.

PoCs for Hybrid Vulnerabilities (15). In this category, traditional DGF tools fail to generate any PoCs since the default options render the target locations unreachable or untriggerable within 24 hours, while ProphetFuzz and ZigZagFuzz generate only 4 and 5 PoCs, respectively. In contrast, CoupleFuzz generates 3.8× and 3.0× more PoCs and achieves speedups of 1.3× and 1.6× on the generated PoCs. On average, CoupleFuzz requires 11.0 hours to generate these PoCs and achieves a speedup of 1.5×. For example, the heap overflow vulnerability tiffcrop-7 [27] requires both a specific, complex set of options (`-E top -F both -I black -R 90 -Z 1:3,1:3 -e separate -k 1 -m -z`) and a specially crafted TIFF file to be triggered. In this challenging case, CoupleFuzz successfully generates a PoC in just 16.0 hours, whereas all traditional DGF and option-aware GF tools fail. This result underscores the superior effectiveness of CoupleFuzz in addressing complex hybrid vulnerabilities that require the coordination between option and file mutations.

PoCs That are Not Generated by CoupleFuzz (4). Although CoupleFuzz is efficient in most cases, there are 4 cases where CoupleFuzz failed to generate (i.e., tiffcrop-1, img2sixel-4, img2sixel-6, and tiffdither-1). To better understand these failures, we conducted an in-depth analysis and identified three key limitations of CoupleFuzz. First, our extraction of options and DIVs depends on Osmart [38], a code-based tool that, while generally effective, may occasionally extract incorrect options and DIVs. Such false positives can negatively affect the OI-fuzzing module's ability to explore relevant input spaces, which in turn led to CoupleFuzz's failure to generate PoCs for tiffdither-1 and img2sixel-6. Second, CoupleFuzz utilizes an incremental mutation strategy to generate option inputs (OIs). While generally effective, it can lead to increased PoC generation time for vulnerabilities that require complex combinations of options. For tiffcrop-1, the triggering command involves 18 options (see our website [5]), versus 9 for tiffcrop-7 above; moreover, many options have continuous and large value spaces (e.g., `-Y`), which prevent energy recycling and lead to a slow convergence. Consequently, CoupleFuzz failed to reach the required combination within the time budget. Third, in cases where vulnerabilities can be triggered using only the default options, CoupleFuzz may spend unnecessary time in the OI-fuzzing module, which can lead to failures in generating PoCs, as observed with img2sixel-4. Nevertheless, this module is crucial for identifying effective option combinations that can significantly accelerate fuzzing in more complex scenarios. The overall efficiency of CoupleFuzz, particularly in the majority hybrid vulnerabilities, demonstrates the value of this module despite its occasional overhead in simpler cases.

Usefulness Evaluation. As discussed in Sec. 1, PoC generation for static vulnerability reports serves two main purposes, reducing the false positive rate of static analysis tools and accelerating vulnerability disclosure, and enhancing the quality of statically detected vulnerabilities. To further

Table 3. PoC Generation Results on Option-Available Dataset ("–" indicates DGF using default options. "*" indicates the generated PoC option by CoupleFuzz is different from expert-provided one. Other notations are consistent with Table 2.)

Program	CVE	DGF				O-GF				O-DGF	
		DAFL		DAFL(–)		ProphetFuzz		ZigZagFuzz		CoupleFuzz	
		Succ.	TTE	Succ.	TTE	Succ.	TTE	Succ.	TTE	Succ.	TTE
swftophp [16]	2018-11095	**10/10**	**0.2**	**10/10**	0.3	0/10	T.O	0/10	T.O	**10/10**	0.8
	2016-9829	**10/10**	**0.7**	**10/10**	0.7	**10/10**	3.1	0/10	T.O	**10/10**	**0.7**
	2017-9988	0/10	T.O	0/10	T.O	9/10	5.0	0/10	T.O	**10/10**	**6.5**
	2017-11728	**10/10**	**0.3**	**10/10**	0.8	**10/10**	1.9	0/10	T.O	**10/10**	0.7
	2017-11729	**10/10**	**0.1**	**10/10**	**0.1**	**10/10**	1.6	0/10	T.O	**10/10**	0.4
	2017-7578	**10/10**	**0.1**	**10/10**	**0.3**	**10/10**	2.1	0/10	T.O	**10/10**	1.1
	2018-11225	0/10	T.O	0/10	T.O	0/10	T.O	0/10	T.O	**2/10**	**17.3**
	2019-12982	9/10	**2.3**	**10/10**	**2.3**	0/10	T.O	0/10	T.O	**10/10**	3.5
	2020-6628	7/10	**11.1**	**8/10**	**10.9**	0/10	T.O	0/10	T.O	6/10	14.3
exiv2 [29, 37]	2017-11339	C.E	C.E	C.E	C.E	**2/10**	**21.2**	0/10	T.O	0/10	T.O
	2017-17669	C.E	C.E	C.E	C.E	C.E	T.O	**1/10**	**23.7**	0/10	T.O
	2018-10780	C.E	C.E	C.E	C.E	C.E	T.O	0/10	T.O	**3/10**	**23.2**
img2sixel [19, 37]	2022-29978	C.E	C.E	C.E	C.E	7/10	10.7*	0/10	T.O	**10/10**	**1.5***
	2019-20094	C.E	C.E	C.E	C.E	0/10	T.O	0/10	T.O	**10/10**	**0.7***
	2022-27046	C.E	C.E	C.E	C.E	0/10	T.O	0/10	T.O	**10/10**	**0.6***
bsdcpio [9, 38]	2015-8915	**10/10**	4.4	**10/10**	8.9	O.E	O.E	0/10	T.O	**10/10**	**3.8**
objcopy [16]	2017-8393	**10/10**	**0.4**	**10/10**	3.2	0/10	T.O	0/10	T.O	**10/10**	0.5
	2017-8394	**10/10**	**0.5**	0/10	T.O	8/10	5.6*	0/10	T.O	**10/10**	**0.5**
	2017-8395	**10/10**	**0.1**	0/10	T.O	0/10	T.O	0/10	T.O	**10/10**	**0.1**
jasper [19, 37]	2016-9398	0/10	T.O	0/10	T.O	O.E	O.E	0/10	T.O	**10/10**	**3.8**
	2018-9252	0/10	T.O	0/10	T.O	O.E	O.E	0/10	T.O	**5/10**	**17.0***
tiffcrop [19, 37]	2023-0801	**10/10**	1.1	0/10	T.O	0/10	T.O	0/10	T.O	**10/10**	**0.7***
	2023-25433	0/10	T.O	0/10	T.O	0/10	T.O	0/10	T.O	**3/10**	**7.2***
	2023-25434	**10/10**	0.9	0/10	T.O	0/10	T.O	0/10	T.O	**10/10**	**0.5***
	2022-2057	0/10	T.O	0/10	T.O	0/10	T.O	6/10	14.7*	**5/10**	**13.4***
	2023-0802	0/10	T.O	0/10	T.O	0/10	T.O	0/10	T.O	**3/10**	**20.0***
	2023-0803	0/10	T.O	0/10	T.O	0/10	T.O	0/10	T.O	**10/10**	**9.1***
	2023-0796	0/10	T.O	0/10	T.O	0/10	T.O	**1/10**	**19.1***	**1/10**	21.9*
	2023-3618	3/10	13.0	0/10	T.O	0/10	T.O	0/10	T.O	**10/10**	**8.2***
	2023-26965	**10/10**	**0.1**	0/10	T.O	**10/10**	3.0*	0/10	T.O	**10/10**	0.2*
strip [16]	2017-7303	**10/10**	**0.2**	**10/10**	1.5	0/10	T.O	0/10	T.O	**10/10**	1.2
objdump [16]	2017-8396	**2/10**	**23.7**	0/10	T.O	0/10	T.O	0/10	T.O	0/10	T.O
	2017-8398	**10/10**	**0.3**	**10/10**	1.1	0/10	T.O	0/10	T.O	**10/10**	0.6
	2017-8392	**10/10**	**0.4**	**10/10**	1.2	**10/10**	6.7	0/10	T.O	**10/10**	0.6
tiffsplit [19, 37]	2016-10095	**10/10**	**6.8**	**10/10**	8.6	0/10	T.O	3/10	18.3	**10/10**	7.7
	2016-9273	**10/10**	**6.8**	**10/10**	8.6	0/10	T.O	0/10	T.O	0/10	T.O
readelf [16]	2017-16828	**10/10**	**0.1**	**10/10**	1.2	0/10	T.O	0/10	T.O	**10/10**	0.3
xmllint [16]	2017-5969	**10/10**	**0.4**	**10/10**	1.6	0/10	T.O	0/10	T.O	**10/10**	1.0
jpeg [3, 37]	2022-31796	0/10	C.E	0/10	C.E	0/10	T.O	**10/10**	5.8	**10/10**	**2.3**
djpeg [3, 37]	2021-29390	0/10	C.E	0/10	C.E	**1/10**	**23.9**	0/10	T.O	0/10	T.O
#Best		20	19	16	5	8	1	3	2	**34**	**18**

assess the practical impact of CoupleFuzz, we examined whether the vulnerabilities corresponding to the static reports for which CoupleFuzz generated PoCs had already been disclosed. After manual verification, we found that 9 PoCs correspond to already disclosed vulnerabilities, while the remaining 13 PoCs are for previously undisclosed issues. For the already disclosed vulnerabilities, CoupleFuzz generated PoCs to enhance the quality of statically detected vulnerabilities by providing concrete evidence of exploitability. For the 13 newly discovered vulnerabilities, we reported both the vulnerabilities and their corresponding PoCs to the respective developers. As a result, 6 of these reports were confirmed by the developers, and 1 received a CVE identifier assignment.

4.2 Efficiency Evaluation on Option-Available Dataset (RQ2)

Traditional DGF datasets typically rely on program options that have been carefully selected and validated by human experts. These expert-provided options are fixed for the duration of fuzzing, allowing DGF tools to operate with high-quality option configurations. Consequently, DGF tools in

these settings focus exclusively on mutating FI, without the need to discover or reason about OI. However, the expert-provided options may not always be the most optimal for the target location. To this end, we investigate whether CoupleFuzz can achieve comparable efficiency through its coordinated OI and FI fuzzing when compared to DGF tools that leverage expert-provided options.

Option-Available Dataset Collection. We selected the dataset of DAFL [16] and filtered (1) 2 cases from the *lrzip* program, as it does not support dynamic taint analysis implemented by DFSan; and (2) 15 cases that could not be reproduced by any baseline within 24 hours. To further enrich the dataset and ensure diversity, we additionally selected projects (e.g., libtiff) that have been used as benchmarks in both at least one DGF work [3, 9, 16, 19, 29] and at least one option-aware GF work [18, 37, 38], ensuring fair comparison with existing baselines. This yields a total of 40 vulnerabilities, with their origins detailed in Table 3.

DGF Setup. To rigorously assess the impact of program options, we evaluate each DGF tool under two configurations: (1) using the expert-provided options, and (2) using the default options. This setup allows us to isolate and quantify the effect of expert-provided options on DGF tools.

Overall Result. As shown in Table 3, CoupleFuzz and DAFL achieve the best performance in 18 (45%) and 19 (47.5%) cases respectively, demonstrating their leadership in the evaluation. In contrast, ProphetFuzz and ZigZagFuzz only achieve the best results in 1 (2.5%) and 2 cases (5.0%), respectively. On average, CoupleFuzz achieves comparable efficiency to DAFL (with a slight speedup of 1.01×), while significantly outperforming ProphetFuzz and ZigZagFuzz with speedups of 3.4× and 1.3×, respectively. Regarding *success ratio*, CoupleFuzz achieves the best results in 34 cases, significantly outperforming DAFL (20 cases). This result is noteworthy; i.e., while DAFL relies on expert-provided options curated for each vulnerability, CoupleFuzz automatically discovers effective option configurations through its OI-fuzzing module. Beyond achieving comparable efficiency without manual effort, CoupleFuzz attains a significantly higher *best success ratio* (34 cases vs. 20 cases), demonstrating that effective PoC generation benefits from the coordinated mutation of both OI and FI. Notably, when DAFL is limited to default options, it fails to generate 7 out of the 21 PoCs that were successfully produced using expert-provided options. After applying the expert-provided options, DAFL shows a speedup of 1.4× for PoC generation. Since neither WindRanger nor AFLGo achieves the best performance in any case, their detailed results are provided at our website [5].

These results rigorously demonstrate that option awareness is crucial for the effectiveness and efficiency of DGF tools. The efficient coordination of option input (OI) and file input (FI) mutation in CoupleFuzz minimizes the overhead of exploring effective option combinations, allowing it to match or surpass the performance of traditional DGF tools that rely on expert-provided options. Thus, CoupleFuzz consistently achieves high effectiveness and efficiency without expert guidance, while traditional DGF tools suffer significant performance drops without such inputs. This underscores the practical value and superiority of option-aware fuzzing tool like CoupleFuzz.

4.3 Ablation Study (RQ3)

To gain deeper insight into the necessity and effectiveness of coordinating FI and OI module, we conduct an ablation study to quantify the individual and combined contributions of each. Specifically, we assess the impact of disabling either the FI or OI module (i.e., w/o FI and w/o OI), as well as the selective instrumentation strategy (i.e., w/o Sele.), as shown in Table 2. To further elucidate the individual contributions of the FI and OI module, we conduct a case study by randomly selecting one vulnerability from each project in the option-unavailable dataset, denoted as (PJ #1-10). For each case, we identify the seed with the shortest average distance to the target location contributed by FI or OI mutations, and repeat this analysis for the w/o FI and w/o OI variants. Here, the distance metric refers to the average executed path distance from the selected

seeds to the target location, as detailed in Sec. 3.7. This experiment enables us to systematically compare how each module influences progress toward the target across different projects.

Ablation Result Analysis. The results in Table 2 rigorously demonstrate the indispensable role of the OI-fuzzing module and FI-fuzzing module. When OI-fuzzing is disabled (w/o OI), PoC generation fails entirely for 21 vulnerabilities. In contrast, for 3 file-dominated vulnerabilities where the default options are already sufficient (e.g., rizin-1), disabling OI-fuzzing leads to a slight reduction in PoC generation time (11.0 vs. 12.7 hours). This suggests that, in scenarios where option exploration is unnecessary, the overhead introduced by OI-fuzzing is minimal and does not hinder efficiency. Thus, OI-fuzzing module is crucial for achieving comprehensive vulnerability coverage, while its impact on performance is negligible when not required.

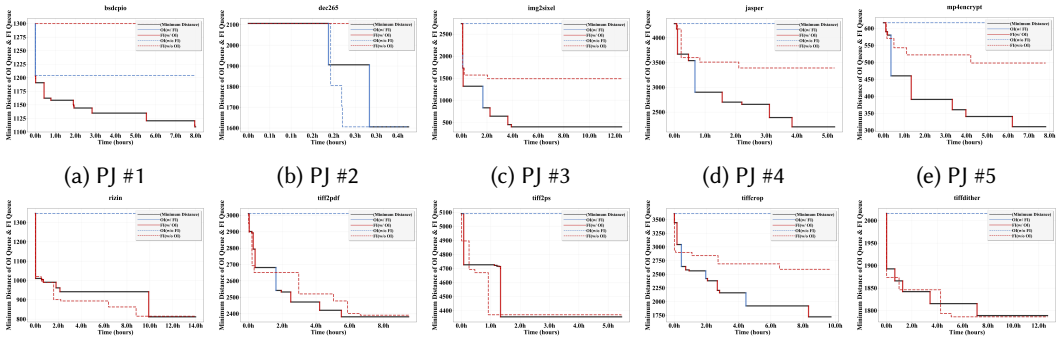
Similarly, removing the FI-fuzzing module (w/o FI) causes a substantial decline in performance, resulting in PoC generation failure for 20 cases of our dataset. This outcome is expected, as the majority of vulnerabilities require mutations to the file input for successful triggering, with the notable exception of purely option-dominated cases such as libde265-1.

Additionally, disabling selective instrumentation (w/o Sele.) leads to a performance degradation and the average TTE increases from 7.4 to 8.2. Moreover, in 3 cases (i.e., mp4encrypt-1, img2sixel-1, and img2sixel-3), the ablated version fails to generate PoC within the time budget where the full CoupleFuzz succeeds. Consistent degradation is also observed on the option-available dataset, where w/o Sele. increases average TTE from 4.0 to 4.8 hours (detailed at website [5]).

Case Study. Fig. 5 shows the runtime of the w/o FI and w/o OI variants compared to the full CoupleFuzz across different projects. The synergy between OI-fuzzing and FI-fuzzing modules is essential for efficiently steering seeds toward the target location. As shown in Fig. 5, vulnerability in PJ #1 exhibits a clear pattern: OI-fuzzing (blue line) first reduces the average distance to the target, after which FI-fuzzing (red line) further advances progress. When either module is disabled, this convergence either slows dramatically or stalls entirely, with no additional basic blocks (BBs) discovered closer to the target. Conversely, for vulnerabilities PJ #3, 4, 5, 7 and 9, FI-fuzzing (red line) initiates the reduction in distance, and OI-fuzzing (blue line) subsequently accelerates convergence. Again, removing either module hinders progress. Notably, in case PJ #2 (option-dominated), omitting FI mutation does not hinder progress while in case PJ #6, 8 and 10 (file-dominated), omitting OI mutation does not hinder progress. Throughout the coordination process, we observe that the relative contributions of OI-fuzzing and FI-fuzzing are not fixed; though FI-fuzzing generally plays a larger role in distance convergence, the contribution from OI-fuzzing remains significant for overall progress, as evidenced by the ablation results. This case study validates our core insight that *cross-guided fuzzing* between OI and FI modules creates a synergistic feedback loop that systematically drives executions toward target locations, demonstrating the practical effectiveness of our coordinated approach in addressing the fundamental challenges of option-aware DGF.

4.4 Threats to Validity

First, our dataset is limited to C/C++ command-line programs and may not generalize to all software benefiting from option-aware DGF. Programs with different option parsing mechanisms (e.g., configuration files) or vulnerabilities with different triggering conditions might exhibit different performance. Second, CoupleFuzz adopts an incremental option exploration strategy that adds at most one option at a time, enabling clear attribution of each option's contribution and avoiding premature pruning. This ensures thorough coverage and reliable identification of effective options, but may slow convergence for vulnerabilities requiring many options simultaneously (e.g., tiffcrop-1). Third, our option-unavailable dataset is derived from CodeQL reports. As LLMs are increasingly adopted for vulnerability detection [20] yet also suffer from high false positive rates [22], integrating CoupleFuzz with LLM-based detectors is a promising direction to broaden its applicability.



(a) PJ #1 (b) PJ #2 (c) PJ #3 (d) PJ #4 (e) PJ #5

(f) PJ #6 (g) PJ #7 (h) PJ #8 (i) PJ #9 (j) PJ #10

Fig. 5. Results for Our Case Study Vulnerabilities PJ #1–10

5 Related Work

Directed Greybox Fuzzing. Generating PoCs for vulnerabilities remains a fundamental challenge. DGF addresses it by guiding test case generation toward specific target locations. AFLGo [2] pioneered this concept by incorporating control-flow distance metrics into the fuzzing process. Hawkeye [3] and WindRanger [9] enhanced AFLGo by providing more precise and informative feedback for computing control-flow distances. Beyond distance-based approaches, several specialized techniques have emerged. ParmeSan [32] incorporates sanitizer-guided bug coverage to improve target reachability. Beacon [13] employs reachability analysis to terminate inputs that cannot reach the target location. Recent advances have focused on selective instrumentation strategies, e.g., SelectFuzz [31] and DAFL [16]. These DGF approaches share a critical limitation, i.e., they ignore the role of program options in determining execution paths. Tofu [39] represents the only existing work that explicitly considers option knowledge in DGF. However, it suffers from the same challenges that plague option-aware GF tools, and it is not publicly available.

Option-Aware Greybox Fuzzing. Current option-aware GF tools lack effective reasoning about which options are relevant for reaching target locations. Document-based tools [17, 36, 37, 43] extract options from documentation, which may be incomplete and irrelevant for target locations. Though code-based tools [38] extract options from source code, they can still include many irrelevant options that cannot influence the semantic toward target locations. As a result, these tools often waste substantial computational resources selecting and mutating ineffective options that have no impact on triggering the vulnerability. Moreover, these tools struggle to effectively coordinate option fuzzing and file fuzzing to target location, leading to inefficient PoC generation.

6 Conclusions

This paper presents CoupleFuzz, a novel option-aware directed greybox fuzzing tool that addresses the critical but under-explored challenge of configurable options in PoC generation. Our evaluation demonstrates that CoupleFuzz significantly outperforms existing DGF tools in PoC generation.

7 Data Availability

All the datasets and code are available at our website [5].

Acknowledgment

This work was supported by the National Natural Science Foundation of China (Grant No. 62332005 and 62372114).

References

[1] 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 *34th USENIX Security Symposium (USENIX Security 25)*. 6977–6997.

[2] 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*. 2329–2344.

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

[4] CodeQL. 2025. *CodeQL*. Retrieved Sep 6, 2025 from https://codeql.github.com/

[5] CoupleFuzz. 2025. *CoupleFuzz*. Retrieved Sep 6, 2025 from https://github.com/optionGo/CoupleFuzz

[6] cve. 2025. *A option-free PoC of libde265*. Retrieved Sep 6, 2025 from https://www.cve.org/CVERecord?id=CVE-2022-43249

[7] Dataflowsanitizer. 2024. *Dataflowsanitizer*. Retrieved Sep 6, 2025 from https://clang.llvm.org/docs/DataFlowSanitizerDesign.html

[8] Peng Deng, Lei Zhang, Yuchuan Meng, Zhemin Yang, and Yuan Zhang. 2025. {ChainFuzz}: Exploiting Upstream Vulnerabilities in {Open-Source} Supply Chains. In *34th USENIX Security Symposium (USENIX Security 25)*. 6199–6218.

[9] Zhengjie Du, Yuekang Li, Yang Liu, and Bing Mao. 2022. Windranger: A directed greybox fuzzer driven by deviation basic blocks. In *Proceedings of the 44th International Conference on Software Engineering*. 2440–2451.

[10] 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)*.

[11] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. 2020. {GREYONE}: Data flow sensitive fuzzing. In *29th USENIX security symposium (USENIX Security 20)*. 2577–2594.

[12] github. 2025. *ruleset of CodeQL*. Retrieved Sep 6, 2025 from https://codeql.github.com/docs/writing-codeql-queries/codeql-queries/

[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 *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 36–50.

[14] Heqing Huang, Peisen Yao, Hung-Chun Chiu, Yiyuan Guo, and Charles Zhang. 2024. Titan: Efficient multi-target directed greybox fuzzing. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1849–1864.

[15] Heqing Huang, Anshunkang Zhou, Mathias Payer, and Charles Zhang. 2024. Everything is good for something: Counterexample-guided directed fuzzing via likely invariant inference. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1956–1973.

[16] Tae Eun Kim, Jaeseung Choi, Kihong Heo, and Sang Kil Cha. 2023. {DAFL}: Directed Grey-box Fuzzing guided by Data Dependency. In *32nd USENIX Security Symposium (USENIX Security 23)*. 4931–4948.

[17] Ahcheong Lee, Irfan Ariq, Yunho Kim, and Moonzoo Kim. 2022. POWER: Program Option-Aware Fuzzer for High Bug Detection Ability.. In *ICST*. 220–231.

[18] Ahcheong Lee, Youngseok Choi, Shin Hong, Yunho Kim, Kyutae Cho, and Moonzoo Kim. 2025. ZigZagFuzz: Interleaved Fuzzing of Program Options and Files. *ACM Transactions on Software Engineering and Methodology* 34, 2 (2025), 1–31.

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

[20] Ahmed Lekssays, Hamza Mouhcine, Khang Tran, Ting Yu, and Issa Khalil. 2025. {LLMxCPG}:{Context-Aware} Vulnerability Detection Through Code Property {Graph-Guided} Large Language Models. In *34th USENIX Security Symposium (USENIX Security 25)*. 489–507.

[21] Penghui Li, Wei Meng, and Chao Zhang. 2024. {SDFuzz}: Target States Driven Directed Fuzzing. In *33rd USENIX Security Symposium (USENIX Security 24)*. 2441–2457.

[22] Yansong Li, Paula Branco, Alexander M Hoole, Manish Marwah, Hari Manassery Koduvely, Guy-Vincent Jourdan, and Stephan Jou. 2025. SV-TrustEval-C: Evaluating Structure and Semantic Reasoning in Large Language Models for Source Code Vulnerability Analysis. In *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE, 3014–3032.

[23] Zongjie Li, Zhibo Liu, Wai Kin Wong, Pingchuan Ma, and Shuai Wang. 2024. Evaluating C/C++ vulnerability detectability of query-based static application security testing tools. *IEEE Transactions on Dependable and Secure Computing* 21, 5 (2024), 4600–4618.

[24] libde265. 2025. *A default option PoC of libde265*. Retrieved Sep 6, 2025 from https://github.com/strukturag/libde265/issues/484

[25] libtiff. 2025. *A buffer overflow Vulnerability in 'combineSeparateSamplesBytes' of libtiff*. Retrieved Sep 6, 2025 from https://gitlab.com/libtiff/libtiff/-/issues/740

[26] libtiff. 2025. *A file dominated PoC of libtiff*. Retrieved Sep 6, 2025 from https://gitlab.com/libtiff/libtiff/-/issues/741

[27] libtiff. 2025. *A hybrid PoC Generated by CoupleFuzz.* Retrieved Sep 6, 2025 from https://gitlab.com/libtiff/libtiff/-/issues/595
[28] libtiff. 2025. *tiffcrop.c.* Retrieved Sep 6, 2025 from https://gitlab.com/libtiff/libtiff/-/blob/master/tools/tiffcrop.c?ref_type=heads
[29] Peihong Lin, Pengfei Wang, Xu Zhou, Wei Xie, Gen Zhang, and Kai Lu. 2024. DeepGo: Predictive Directed Greybox Fuzzing. In *Proceedings of the 31st Network and Distributed System Security Symposium (NDSS)*.
[30] American Fuzzy Lop. 2024. *American Fuzzy Lop.* Retrieved Sep 6, 2025 from https://github.com/google/AFL
[31] Changhua Luo, Wei Meng, and Penghui Li. 2023. Selectfuzz: Efficient directed fuzzing with selective path exploration. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2693–2707.
[32] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2020. {ParmeSan}: Sanitizer-guided greybox fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*. 2289–2306.
[33] Huanyao Rong, Wei You, Xiaofeng Wang, and Tianhao Mao. 2024. Toward Unbiased {Multiple-Target} Fuzzing with Path Diversity. In *33rd USENIX Security Symposium (USENIX Security 24)*. 2475–2492.
[34] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th international conference on compiler construction*. 265–266.
[35] Tamás Szabó. 2023. Incrementalizing production codeql analyses. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1716–1726.
[36] Dawei Wang, Ying Li, Zhiyu Zhang, and Kai Chen. 2023. {CarpetFuzz}: Automatic program option constraint extraction from documentation for fuzzing. In *32nd USENIX Security Symposium (USENIX Security 23)*. 1919–1936.
[37] 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 on ACM SIGSAC Conference on Computer and Communications Security*. 735–749.
[38] 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 on ACM SIGSAC Conference on Computer and Communications Security*. 705–719.
[39] Zi Wang, Ben Liblit, and Thomas Reps. 2020. Tofu: Target-oriented fuzzer. *arXiv preprint arXiv:2004.14375* (2020).
[40] 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 *33rd USENIX Security Symposium (USENIX Security 24)*. 2459–2474.
[41] Songtao Yang, Yubo He, Kaixiang Chen, Zheyu Ma, Xiapu Luo, Yong Xie, Jianjun Chen, and Chao Zhang. 2023. 1dfuzz: Reproduce 1-day vulnerabilities with directed differential fuzzing. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 867–879.
[42] Yujian Zhang, Yaokun Liu, Jinyu Xu, and Yanhao Wang. 2024. Predecessor-aware directed greybox fuzzing. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1884–1900.
[43] Zenong Zhang, George Klees, Eric Wang, Michael Hicks, and Shiyi Wei. 2022. Registered report: Fuzzing configurations of program options. In *San Diego, CA, USA: International Fuzzing Workshop (FUZZING)*.
[44] Han Zheng, Jiayuan Zhang, Yuhang Huang, Zezhong Ren, He Wang, Chunjie Cao, Yuqing Zhang, Flavio Toffalini, and Mathias Payer. 2023. {FISHFUZZ}: Catch deeper bugs by throwing larger nets. In *32nd USENIX Security Symposium (USENIX Security 23)*. 1343–1360.
[45] Xin Zhou, Sicong Cao, Xiaobing Sun, and David Lo. 2025. Large language model for vulnerability detection and repair: Literature review and the road ahead. *ACM Transactions on Software Engineering and Methodology* 34, 5 (2025), 1–31.
[46] Peiyuan Zong, Tao Lv, Dawei Wang, Zizhuang Deng, Ruigang Liang, and Kai Chen. 2020. {FuzzGuard}: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning. In *29th USENIX security symposium (USENIX security 20)*. 2255–2269.

Received 2026-02-23; accepted 2026-03-24