Context-Free Property Oriented Fuzzing

Jiaqiang Yao[*] [†]
College of Computer Science and Technology, National University of Defense Technology
Changsha, China
jiaqiangyao@nudt.edu.cn

Meixi Liu[*] [†]
College of Computer Science and Technology, National University of Defense Technology
Changsha, China
liumeixi@nudt.edu.cn

Zhenbang Chen[†] [‡]
College of Computer Science and Technology, National University of Defense Technology
Changsha, China
zbchen@nudt.edu.cn

Yongchao Xing[†]
College of Computer Science and Technology, National University of Defense Technology
Changsha, China
xingyc0979@nudt.edu.cn

Jinjian Luo[†]
College of Computer Science and Technology, National University of Defense Technology
Changsha, China
luojinjian@nudt.edu.cn

Yunlai Luo[†]
College of Computer Science and Technology, National University of Defense Technology
Changsha, China
luoyl@nudt.edu.cn

Guofeng Zhang[†]
College of Computer Science and Technology, National University of Defense Technology
Changsha, China
zhangguofeng16@nudt.edu.cn

Yufeng Zhang
College of Computer Science and Electronic Engineerin, Hunan University
Changsha, China
yufengzhang@hnu.edu.cn

Ji Wang[†]
College of Computer Science and Technology, National University of Defense Technology
Changsha, China
wj@nudt.edu.cn

Abstract

Fuzzing is effective for finding software bugs. However, the bugs specified in context-free properties are difficult for the existing fuzzers. These bugs are triggered when the program execution contains specific sequences of operations, *e.g.*, push and pop operations on the stack, and locking and unlocking operations on the lock. As far as we know, existing approaches do not support fuzzing for *non-regular* context-free properties, which are more expressive and can be used to specify bugs in many scenarios.

This paper proposes a general runtime monitoring-based fuzzing framework for the bugs expressed as context-free properties. We propose two algorithms to improve fuzzing's effectiveness and efficiency with respect to the context-free property. The algorithm for preserving input mutants leverages the state transition information of the property's monitors. The other algorithm for mutating the input seed combines control flow information with state transition information to prioritize the different parts of the input. We have implemented our framework CFPOFuzz for C/C++ programs. The results of the extensive experiments on real-world C/C++ programs indicate our method's effectiveness and efficiency. Compared with coverage-oriented fuzzing, our method achieves 3.83x speedups for generating the first target input triggering the bugs. Our method found 7 unknown zero-day bugs.

[*]The first two authors contributed equally to this research and are co-first authors.
[†]Also with the affiliation: State Key Laboratory of Complex & Critical Software Environment, National University of Defense Technology, Changsha, Hunan
[‡]Zhenbang Chen is the corresponding author.

CCS Concepts

• **Software and its engineering** → **Software testing and debugging**; • **Security and privacy** → **Software and application security**.

Keywords

Fuzzing, Context-free grammar, Pushdown automata

ACM Reference Format:
Jiaqiang Yao, Meixi Liu, Zhenbang Chen, Yongchao Xing, Jinjian Luo, Yunlai Luo, Guofeng Zhang, Yufeng Zhang, and Ji Wang. 2026. Context-Free Property Oriented Fuzzing. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3744916.3773273

1 Introduction

Fuzzing [26, 41] is an effective method for automatically testing software. It generates program inputs by mutating the initial inputs (or seeds) and then feeds the mutated inputs to the program to catch program bugs. Recently, there have been significant advancements [1, 33, 40] in fuzzing techniques, which generalizes fuzzing's application scope and improves scalability. Fuzzing has uncovered numerous previously unknown vulnerabilities [30, 40] and is widely adopted in industry [1] to enhance software reliability.

In principle, program properties can specify bugs [22]. We refer to properties that can be formally specified by a context-free grammar [12] or a pushdown automaton (PDA) [13] as context-free properties. For real-world programs, many properties (or bugs) require the expressiveness of context-free properties. There are many application-independent context-free properties. For instance, the number of stack pop operations should not exceed the number of push operations, and the equality of unlock and lock operations. Moreover, accurately specifying the root causes of many bugs often requires context-free languages. To precisely define these root causes, we may need implementation- or functionality-related information; then, we often need context-free properties. For example, although CVE-2021-23134 in the Linux kernel is a use-after-free (UAF) bug [5], the root cause is the mismatch between a memory object's reference count and the number of the object's references, where the object will be automatically freed when its reference count is zero, which may cause a UAF bug when the object is already freed but the references of the object exist, and the object is dereferenced later. The specification of this root cause needs non-regular context-free properties. Additionally, for many protocol implementations, the number of handshake messages should never be fewer than the number of authentication messages, which may cause unintended behaviors, such as denial of service. Thus, using context-free properties enhances the precision and scope of specifying bug properties, thereby promoting bug detection.

Ensuring that these context-free properties are not violated or left unsatisfied at any time is crucial for ensuring the security of software systems. Triggering these bugs often requires not only a specific sequence of operations but also quantitative relationships among them, such as equality, inequality, or other constraints. Unfortunately, those properties are challenging to analyze or verify due to the complexity of specification and verification. In principle, the complexity of verifying a context-free property for a pushdown system is *undecidable* [12]. Few approaches exist for analyzing or verifying a program's context-free properties.

Currently, many mainstream fuzzers [1, 33, 40] are program coverage-oriented. The key idea is to improve the program coverage (*e.g.*, statement coverage or branch coverage [27]) as soon as possible, which aims to find bugs as many as possible. Coverage-guided fuzzing [40] are good at finding reachability bugs, such as assertion violation [25], division by zero, and array index out-of-bound. However, the bug specified by a context-free property (context-free bug) imposes both temporal and quantitative constraints on program execution. Satisfying coverage requirements is inefficient and may even fail to detect context-free bugs (demonstrated in Section 2).

There are some approaches for fuzzing specific temporal bugs [28, 29, 37, 38], which suffer from the generalizability problem. The framework in [37] is general for fuzzing typestate [35] bugs by imposing a typestate-oriented static analysis guided fuzzing. Due to the imprecision problem of static analysis, this approach may produce false alarms if we apply it to general typestate properties. LTLFuzzer [24] aims to improve fuzzing for LTL properties by synergizing runtime verification and model checking with gray box fuzzing. But none of them can express the aforementioned context-free properties, which are more expressive than regular or typestate properties specified by Finite State Machine (FSM) or Linear-time Temporal Logic (LTL). *As far as we know, none of the existing work supports the fuzzing for non-regular context-free properties, which are more expressive and needed for specifying the bugs of many scenarios.*

In this paper, we propose a general fuzzing framework for context-free properties. More specifically, we consider *non-regular context-free properties.* Given a context-free property ψ and a program $\mathcal{P}$, our framework automatically generates a runtime monitor [20] with respect to ψ. The monitor is implemented as a pushdown automaton that checks whether the execution can satisfy ψ. The monitor code will then be automatically woven into $\mathcal{P}$ to monitor $\mathcal{P}$'s execution and *precisely* inspect $\mathcal{P}$'s behavior. To improve our fuzzing framework's efficiency, we improve the *mutant preservation* and *input mutation* in fuzzing based on the following two key insights. The first one is that different program inputs have a different relevance to ψ. For example, suppose that an input I can trigger more state transitions of ψ's monitor and reach the state closer to the automata's accepting states. In that case, it is *more possible* to mutate I to get the inputs whose program executions satisfy ψ. Besides, different input parts have different influences for the executions satisfying ψ. Therefore, we need to mutate the parts more related to ψ, *i.e.*, whose mutations are more likely to generate $\mathcal{P}$'s executions satisfying ψ.

These insights inspire us to propose two novel algorithms to improve fuzzing's efficiency. Based on the first insight, we propose a fuzzing algorithm that monitors $\mathcal{P}$'s executions with respect to ψ and keeps the mutants that trigger more transitions or reach the states that are closer to ψ's monitor's accepting state. Then, based on the kept mutants, the framework can continue fuzzing to find bugs. The second insight inspires us to propose an algorithm to prioritize the input parts with respect to their possibilities of making transitions towards the accepting state, *i.e.*, triggering the bugs specified by ψ. Our algorithm combines the information of control flows [2] and the state transitions with respect to ψ to prioritize the input's different parts. Then, our fuzzing framework allocates more mutation opportunities to higher-priority input parts. We have implemented our fuzzing framework CFPOFuzz for C/C++ programs. The results of the extensive experiments on real-world programs demonstrate our method's effectiveness and efficiency.

The main contributions of this paper are as follows.

- We propose a context-free property-guided fuzzing framework for detecting temporal bugs. Our framework does not produce false alarms and detects all the context-free bugs possibly triggered by the input. As far as we know, we are the first fuzzing framework that supports non-regular context-free properties.
- We propose a mutant preservation algorithm that keeps the ones resulting in more transitions and closer to triggering the bugs.
- We propose an input prioritization algorithm that selects the more context-free property bug-related input parts for mutation.
- We have implemented our fuzzing method CFPOFuzz for C/C++ programs. We evaluate CFPOFuzz on real-world programs and their representative context-free properties. The experimental results indicate that CFPOFuzz achieves 3.83x speedups for detecting CVEs compared with the baselines. Besides, CFPOFuzz detects 7 unknown zero-day bugs.

2 Illustration

This section first introduces context-free properties. Then, we use a motivation example to demonstrate our method.

2.1 Context-free property

A temporal property ψ of a program $\mathcal{P}$ gives an abstraction model of $\mathcal{P}$. ψ can be specified in many forms, such as temporal logic [22],

$$S \longrightarrow \text{push } S \mid \text{push } S \text{ pop} \mid S\, S \mid \epsilon$$

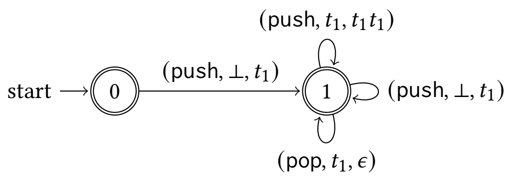

Figure 1: A context-free property's grammar and pushdown automata.

grammar [13], and automata [13]. This paper considers context-free properties, which can be specified by context-free grammars [12] defined as follows.

Definition 2.1. (Context-free grammar) A context-free grammar (CGF) $\mathcal{G}$ is a 4-tuple (Σ, NT, P, S), where

- Σ and NT are the sets of *terminals* and *non-terminals*, respectively.
- $P \subseteq NT \times (NT \cup \Sigma)^*$ is the set of production rules, where $(NT \cup \Sigma)^*$ represents the language (including the empty word ϵ) of each word which is composed of the elements from $NT \cup \Sigma$. Each production rule (nt, α) in P represents that the non-terminal nt can be rewritten by the word α in the production, and we use $nt \longrightarrow \alpha$ to represent the rule.
- $S \in NT$ is the start non-terminal.

In the following, we call the bugs specified in context-free properties *context-free bugs*. Context-free grammars and pushdown automata have the same expressiveness and can be converted to each other [12]. This paper uses context-free grammars for specification and their corresponding pushdown automata for runtime monitoring. A pushdown automaton is defined as follows.

Definition 2.2. (Pushdown automata) A pushdown automaton (PDA) $\mathcal{M}$ is a tuple $(\Sigma, \Gamma, Q, q_0, \delta, \mathcal{A})$, where

- Σ and Γ are the set of inputs and stack alphabets, respectively, and $\perp \in \Gamma$ is the special bottom-of-stack symbol.
- Q is a finite state set, $q_0 \in Q$ is the initial state, and $\mathcal{A} \subseteq Q$ is the accepting state set.
- $\delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \times Q \times \Gamma^*$ is the state transition set. Each transition (q_1, a, t, q_2, α) represents that the automaton can transit to state q_2 from state q_1 when the input is a and the stack's top is t, after which the stack's top t will be popped, and α will be pushed into the stack.

Context-free grammars and PDAs are more expressive than regular grammars and LTL. Furthermore, regular properties (regular grammars or FSMs) and LTL are memoryless models that cannot express the non-regular context-free properties. For example, the language $\{a^n b^n \mid n \geq 0\}$ can be specified by a PDA but not by an FSM or an LTL.

Figure 1 shows a property ψ for the `Stack` class, which requires that the popping times are less or equal to the pushing times at any time. Figure 1 gives ψ's CFG and PDA. State 0 is the initial state, and both states are accepting states. There are two elements in the alphabet for this property. *In the following, we use the event to refer to the alphabet.* Event push corresponds to the invocation of push method of a stack object. Event pop corresponds to invoking the object's pop method, which pops the top element from the stack

```
1  int main() {
2      char input[7];
3      scanf("%6s", input);
4      stack<int> s;
5      int i = 0;
6      s.push(1);
7      int x = rand() + 100;
8      for(int j = 0;j <= x;j++){
9          s.push(j);
10         s.pop();
11     }
12     if (input[0] == 'a')
13         i = i + 1;
14     if (input[4] == 'b') {
15         if (input[1] == 'c')
16             s.pop();
17     }
18     if (input[5] == 'd') {
19         if (input[2] == 'e')
20             s.pop();
21         if (input[3] == 'f')
22             s.push(i);
23     }
24     return 0;
25 }
```

Figure 2: A motivating example program.

object. Hence, $\Sigma = \{\text{push}, \text{pop}\}$. There is only one stack symbol, t_1, representing that the stack is not empty. The triplet on each edge consists of the accepted event, the stack's top element, and the pushed elements. Hence, the automaton pushes t_1 into the stack when accepting event push. When the stack is not empty, *i.e.*, the top element is t_1, the automaton can accept event pop, after which the automaton pops t_1 from the stack and pushes nothing. When ψ's PDA does not accept the event sequence, a violation happens, *i.e.*, popping more times than pushing.

2.2 Motivation Example

This subsection gives an example to demonstrate our fuzzing framework. Figure 2 shows a C++ program (denoted by $\mathcal{P}$) containing stack operations. There is a context-free bug in $\mathcal{P}$ that can be triggered by the input aceabd, which violates the stack safety property illustrated in Figure 1. Next, we first present the results of applying coverage-guided fuzzing to $\mathcal{P}$, followed by our method.

2.2.1 Coverage-oriented fuzzing. Suppose that we have instrumented $\mathcal{P}$ to enable the runtime monitoring [14], which will report the context-free bug if one of bug property ψ's accepting states is reached during fuzzing. If we use coverage-oriented fuzzing, we start with a set of initial seeds. Then, we select one seed from the seed pool and randomly mutate the seed to generate mutant inputs, which will then be used to execute $\mathcal{P}$. The mutants that increase the coverage will be kept and added to the seed pool. This loop continues until the coverage requirement is satisfied or timeout. Suppose we use branch coverage [27] as the coverage criterion and the initial seed is hhhhhh, and the coverage-based fuzzing generates the following mutant inputs.

- ahhhhh covers $12 \longrightarrow 13 \longrightarrow 14 \longrightarrow 18$.
- hhhhhd covers $12 \longrightarrow 14 \longrightarrow 18 \longrightarrow 19 \longrightarrow 21$.
- hhehhd covers $12 \longrightarrow 14 \longrightarrow 18 \longrightarrow 19 \longrightarrow 20 \longrightarrow 21$.

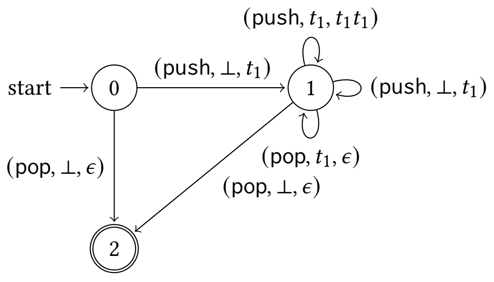

Figure 3: The runtime monitor of motivation example.

- hhefhd covers 12 → 14 → 18 → 19 → 20 → 21 → 22.
- hhhhbh covers 12 → 14 → 15 → 18.
- hchhbh covers 12 → 14 → 15 → 16 → 18.

Then, all the branches of $\mathcal{P}$ are covered. However, the bug is not triggered, which needs the inputs whose second, third, fifth, and sixth characters are c, e, b, and d, respectively, *e.g.*, hcehbd. The reason is that coverage-oriented fuzzing only considers branch coverage instead of the statement sequence requirements of the context-free bug. *When we use AFL for fuzzing the program $\mathcal{P}$ with respect to branch coverage, AFL fails to generate the input triggering the exception in 10 minutes.*

2.2.2 *Context-free property oriented fuzzing.* Now, we demonstrate our context-free property-oriented fuzzing by fuzzing $\mathcal{P}$ with respect to bug property ψ. It should be noted that both our instrumentation and monitoring are based on the bug property ψ as shown in Figure 3, rather than the safety stack property for correct program execution presented in Figure 1.

When there are more pop events than push events in the event sequence, the monitor will enter the accepting state 2 (Figure 3), *e.g.*, ⟨push, pop, pop⟩. Before fuzzing, we instrument $\mathcal{P}$ to enable runtime monitoring and mutation prioritization (Section 3.2.2). Specifically, we add the instrumentations before Lines 6, 9, 10, 16, 20, and 22. The instrumentations generate the events at runtime to advance the monitor with respect to the PDA in Figure 3. Besides, we also instrument before the conditions at Lines 8, 14, 15, 18, 19, and 21 to collect the control flow information and the possible state transition information. For example, before Line 14, we add the instrumentation that reports the value `input[4]`− 'b' [16], the current state of the runtime monitor, and the events contained in the condition's branch. The first value calculates the relation between each input byte and the condition; the last two pieces of information are used to calculate the possibility of mutating a byte to trigger a state transition. Note that we do not need to instrument before the condition at Line 12 because the condition does not control any event statements related to ψ.

Then, when fuzzing starts, we utilize the information collected by the instrumentations to improve fuzzing's efficiency as follows. Like coverage-oriented fuzzing, we suppose that the initial seed is hhhhhh. Then, we mutate each byte of the initial seed a few times and calculate the priorities of the input's bytes. Because `input[0]` is not related to any event statements, *i.e.*, calculated from the information reported by the instrumentations based on information entropy calculation [21], `input[0]` is given a lower priority. Besides, `input[1]`, `input[2]` and `input[3]` will also be given a lower priority because only mutating one byte cannot reach the conditions that they are related to. Different from these four bytes, `input[4]` and `input[5]` will be prioritized because the execution of their mutations indicates that these two bytes are related to the conditions at Lines 14&18, which also control the event statements of ψ.

Suppose that we select `input[4]` for mutation and the generated mutants contain hhhhbh, which covers a new branch. This mutant will be added to the seed pool for the later fuzzing. Then, we select hhhhbh in the next fuzzing iteration. Same as before, we need to prioritize the bytes of the input. The result is that `input[1]` and `input[5]` are prioritized in the same priority, and the reason is similar to the last fuzzing iteration. Suppose we choose `input[1]` for mutation, and the preserved mutant is hchhbh because this mutant results in a new transition according to the PDA in Figure 3. Based on this mutant, we prioritize `input[5]` in the third fuzzing iteration and generate hchhbd to satisfy the condition at Line 18.

Then, in the next iteration, hchhbd is selected as the seed for mutation. The execution preceding the conditions at Lines 19&21 produces the event sequence $\langle \text{push}, \text{pop} \rangle^*$ (a sequence of multiple push-pop operation pairs), causing the PDA to reach state 1 with an empty stack. Both true branches contain events relevant to ψ, but the condition at Line 22 includes only events that increase the distance to the accepting states. As a result, `input[2]` is assigned higher priority than `input[3]` and is selected in priority for mutation. which generates the mutant to satisfy the condition at Line 19, *e.g.*, hcehbd, to trigger the context-free bug.

In summary, *compared with coverage-oriented fuzzing, our method only needs four fuzzing iterations to generate the input triggering the bug. Our implementation needs 34.6s to trigger the bug.*

Note that when using FSM or LTL-bounded approximations of context-free properties (e.g., limiting matched sequences to $n \leq 10$) may detect some real violations. However, relying solely on such approximations can lead to both false negatives (missing deeper bugs) and false positives (misclassifying correct behaviors). Moreover, in program $\mathcal{P}$, the context-free bug is triggered only after hundreds of event matches, with the triggering event located deep in the sequence, making FSM or LTL approximations infeasible. Consequently, existing approaches such as UAFL [37] and LTLFuzzer [24] cannot approximate the context-free properties targeted in this work.

3 Context-Free Property Oriented Fuzzing

This section shows the details of our framework for fuzzing context-free bugs.

3.1 Framework

To support fuzzing general context-free bugs and improve the scalability, we propose the fuzzing framework in Figure 4. Our framework can be divided into two stages. The first stage instruments the program with respect to the context-free bug property. The second stage carries out the fuzzing loop and utilizes the information produced by the instrumentations to improve the fuzzing's efficiency. Next, we explain the key steps in each stage. Suppose the program is $\mathcal{P}$ and the bug property is ψ.

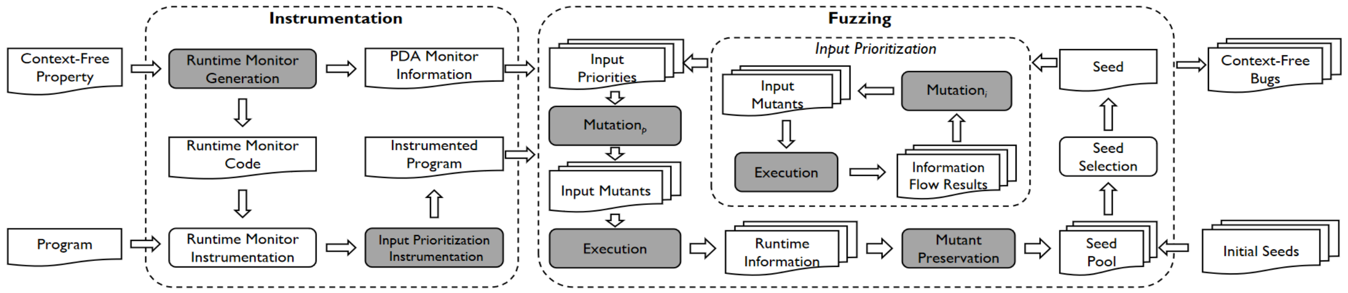

Figure 4: Context-free property oriented fuzzing framework.

```
Stack(void* key) {
    event push before(void* key):
        (call(% push(...))) && target(key){}
    event pop before(void* key):
        (call(% pop(...))) && target(key){}
    cfg :
        S -> push S pop | S S | push S | epsilon
    @fail{
        printf("Errs at %p!\n",key);
        RESET;
    }
}
```

Figure 5: Buggy Stack Context-Free Property.

The first stage contains the following key steps: runtime monitor generation, runtime monitor instrumentation, and input prioritization instrumentation. The second stage is mainly a traditional fuzzing loop. There are the following steps to explain: seed preservation, input prioritization, and seed energy.

The fuzzing loop continues until the time is up. Inputs that trigger the context-free bug, *i.e.*, those that cause ψ to reach the accepting states, are recorded and reported after fuzzing.

3.2 Instrumentation

3.2.1 Context-Free Property. The properties are customized according to different testing requirements, specifications, concerned events, and required instrumentation. The specifications are event sequence constraints described by a context-free grammar.

Figure 5 illustrates the bug property of a stack. It defines two events, "push" and "pop", and specifies instrumentation rules that map events to program behaviors. For example, "push" corresponds to the function call "stack.push(...)". The monitor is parameter-sensitive, capturing the target object of each call and using it as the parameter for the corresponding "push" or "pop" event. The specification defined by the context-free grammar ensures that, for any stack instance, pushes must never be fewer than pops(a safety specification). We then use **@fail** to monitor its violation. That is, the monitor is implemented based on the stack's bug property, with the corresponding PDA shown in Figure 3. Conversely, **@match** indicates that when the specification expressed by the CFG is validated.

3.2.2 Runtime Monitoring. Given a context-free property ψ, we perform the instrumentation and compilation of the program. Our instrumentation process is divided into three parts: Runtime monitor instrumentation, prioritization instrumentation (associated with events), and original coverage instrumentation.

Runtime monitor generation. Based on the defined property ψ, the corresponding monitoring code is generated *automatically*. Usually, a monitor implements a PDA to monitor $\mathcal{P}$'s execution. Besides, the monitor must also send the state transition information to the fuzzer for efficient fuzzing.

Runtime monitor instrumentation. Runtime monitor instrumentation is implemented using AOP-based techniques [15] at the abstract syntax tree (AST) level, weaving monitoring code into $\mathcal{P}$ at compile time. Specifically, the instrumentations are added around the related statements of ψ's events. These instrumentations generate the corresponding events to trigger the transitions of the runtime monitors. Then the instrumented AST is subsequently compiled into LLVM IR, which serves as the input to the prioritization instrumentation.

Input prioritization instrumentation. Different input parts have different influences for detecting ψ's context-free bug. To prioritize the different parts of the input, we propose a method combining the entropy-based information flow calculation [21, 23] and the context-free bug-oriented possibility calculation, which considers both the satisfaction of the branch conditions controlling ψ's events and the possibility of the state transitions making ψ to be satisfied. This method needs to add some instrumentations to the related condition statements, and the instrumentations are in charge of collecting the control flow information and ψ's state transition information during fuzzing.

Finally, the LLVM IR with prioritization instrumentation is compiled into binary, and coverage instrumentation is applied at the binary level.

Monitoring at Runtime. Then, at runtime, we create a runtime monitor for each runtime object whose class or type is specified by ψ. In principle, each runtime monitor is a pair (o, s), where o represents the runtime object's identity, and s represents o's current state in ψ. Hence, runtime monitoring is *object parametric.* Note that ψ may involve multiple objects where the runtime monitors contain a tuple of object identities. *Our framework and implementation support the properties of multiple objects,* e.g.,*different stack objects in Figure 5.* For brevity, we omit the formal definition of the runtime monitors involving multiple objects.

We use $\mathcal{P}_I$ to denote the program after instrumentation, and $\mathcal{P}_I$ will be used for fuzzing. When executing $\mathcal{P}_I$, if the instrumentation generates an event e related to an object o, the monitoring code first checks whether a runtime monitor exists for o. If the monitor does not exist, we create a runtime monitor (o, s_0), where s_0 is ψ's initial state. Then, we get o's runtime monitor (o, s_c) and do the state transition caused by e with respect to ψ. Our approach supports the properties of deterministic pushdown automaton. The monitor is implemented based on the traditional shift-reduce algorithm for LR(1). When multiple choices exist in the shift-reduce table, we clone the stack and merge the equivalent stacks to reduce the overhead. If event e can advance s_c to s_n, o's runtime monitor will be updated to (o, s_n). This way, the runtime monitors are created and updated along with $\mathcal{P}_I$'s execution.

For example, for Figure 2's program, there will be an instrumentation before the statement at Line 6, and the instrumentation generates an event push. Before executing the statement, the instrumentation creates a runtime monitor $(o_t, 0)$ first, where o_t is the identity of the stack object created at Line 4 and state 0 is ψ's initial state. Then, the monitor will be updated to $(o_t, 1)$ by the event push. If the input is "hcehbd", *i.e.*, the input triggering the bug, the generated event sequence is $\langle$push, pop, pop$\rangle$, which will change the monitor to $(o_t, 2)$, where state 2 is ψ's accepting states. A bug is detected.

3.3 Fuzzing

The fuzzing loop is based on the instrumented program under test $\mathcal{P}_I$, the bug property's PDA monitor, and a set of initial seed inputs for $\mathcal{P}_I$. In the beginning, all the initial input seeds are added to the seed pool. We also maintain the global maximum times T of the state transitions in one execution and the shortest distance D to the accepting states of ψ. Then, the fuzzing procedure continues until timing out (omitted for brevity).

Seed Selection. In each fuzzing iteration, a seed is selected from the seed pool. Seed selection can adopt various strategies. We use the round-robin style [4] to select the seed. The input is then prioritized via input prioritization, with details provided in 3.4.

Mutation. Next, we mutate the seed s with respect to the input priorities, *i.e.*, Mutation$_p$. Energy(s) determines the number of mutants generated (see Section 3.5). Mutate$_p$ mutates s for a few rounds to generate an input mutant. Each round mutates a random-sized continuous part of s (denoted by R) starting from a randomly generated offset. Then, for each part p, Mutate$_p$ determines whether to mutate p with respect to p's priority. Now, the mutation probability of the part with the highest priority in R is 100%; the probability of the part with zero priority is 30%; the probability of the remaining parts is 65%.

To mitigate the risk of heuristic prioritization overlooking potential bugs in non-prioritized input regions, we retain mutation on these segments. Regions with the lowest priority are still assigned a minimum mutation probability of 30%, ensuring systematic exploration of latent bugs across the entire input space.

Execution. Next, we execute $\mathcal{P}_I$ under mutant. The execution of $\mathcal{P}_I$ will return the number of state transitions and the shortest distance to ψ's accepting states, which are collected by the instrumentations added at 3.2. Then, we update the global T and D if the mutant is better, *i.e.*, resulting in more transitions or a shorter distance to the accepting states.

Algorithm 1: Context-free Property Oriented Mutant Preservation

```
Preserve($\mathcal{P}_I, S_p, m, \mathcal{T}, \mathcal{D}$)
Data: $\mathcal{P}_I$ is an instrumented program, $S_p$ is the seed pool, m is the mutant,
      and $\mathcal{T}$ and $\mathcal{D}$ are the global maximum number of transitions and the
      global shortest distance to the accepting state on the bug's PDA monitor
1  begin
2      $(T_m, D_m) \leftarrow$ Execute($\mathcal{P}_I, m$)          ▷ m is generated by mutating s
3                                  ▷ $D_m$ is calculated on-the-fly based on the Algorithm 2
4      if $T_m > \mathcal{T}$ then
5          $\mathcal{T} \leftarrow T_m$
6          $S_p \leftarrow S_p \cup \{m\}$          ▷ Increased number of state transitions
7      end
8      else if $D_m < \mathcal{D}$ then
9          $\mathcal{D} \leftarrow D_m$
10         $S_p \leftarrow S_p \cup \{m\}$          ▷ Shorter distance to the accepting state
11     end
12     else if $T_m = \mathcal{T} \wedge D_m = \mathcal{D} \wedge m$ improves coverage then
13         $S_p \leftarrow S_p \cup \{m\}$          ▷ Optimal seeds from different paths
14     end
15 end
```

Mutant preservation. Algorithm 1 shows the details of our mutant preservation. The algorithm executes $\mathcal{P}_I$ with each mutant m. We keep m if one of the following cases holds.

- The number of state transitions triggered by executing $\mathcal{P}_I$ with m exceeds the global maximum T (Lines 4-7), because more transitions indicate that the execution of the program has triggered more events, containing richer execution information, which tends to advance ψ's PDA to the accepting states. The bug cannot be revealed if there are no state transitions when executing m.
- The shortest distance to ψ's accepting states is less than the global shortest distance D (Lines 8-11), which means that m is closer to triggering the bug.
- If executing $\mathcal{P}_I$ with m results in the same number of transitions and the same distance to the accepting states, we keep m when it improves the coverage (Lines 12-14).

The kept mutant is considered a promising candidate to trigger the bug and added to the seed pool S_p for later fuzzing. Besides, if the execution of a mutant triggers the bug, *i.e.*, the shortest distance D_m is zero, the framework records the mutant and continues the fuzzing, which is also omitted for brevity.

Calculation of Distance to Accepting States. For *non-regular* context-free properties, the monitor is implemented as a PDA $\mathcal{M}$. It is inaccurate to statically calculate the distances to the accepting states based on $\mathcal{M}$'s state transitions because the transitions depend on $\mathcal{M}$'s stack state and the input event. To improve the precision of distance calculation, we calculate the distance to the accepting states *on-the-fly*, as shown in Algorithm 2. Specifically, we calculate state's distance to the accepting states according to the current state and $\mathcal{M}$'s current stack, which is known at runtime. We try all the possible events up to length Q (Line 9). For each event, we simulate the execution of the event according to current state and $\mathcal{M}$'s definitions, including the pushing and popping operations to the stack (Line 10). If there exist event sequences that can make $\mathcal{M}$ transit to an accepting state, we use the length of the event sequence

Algorithm 2: Calculation of the Distance to the Accepting States on the PDA

```
DistanceCalculate($\mathcal{M}, Q, State, E$)
Data: $\mathcal{M}$ is the corresponding PDA, $Q$ is the distance limit, State is the
      current state of the PDA, E is the set of events.
begin
    if State is an accepting state on $\mathcal{M}$ then
        return 0
    end
    $QS \leftarrow \emptyset$    ▷ $QS$ is an queue with the element (Stack, State, Distance)
    Enqueue($QS$, ($\mathcal{M}.stack, State, 0$))    ▷ Enqueue an element
    while $QS$ is not empty do
        ($DFAstack, S, Dis$) ← Dequeue($QS$)    ▷ Dequeue an element
        for $e$ in $E$ do
            ($DFAstack_{new}, S_{new}$)←Simulate($\mathcal{M}, DFAstack, S, e$)
            $Dis_{new} = Dis + 1$
            if $S_{new}$ is an accepting state on $\mathcal{M}$ then
                return $Dis_{new}$
            end
            else if $Dis_{new} < Q$ then
                Enqueue($QS$, ($DFAstack_{new}, S_{new}, Dis_{new}$))
            end
        end
    end
    return Q    ▷ The distance to the accepting state exceeds Q
end
```

with the minimum length as the distance (Lines 13); Otherwise, we use Q as the distance (Lines 20).

Distance computation incurs additional runtime overhead. However, in practical software security scenarios, most properties can be abstracted as simple context-free patterns with alternating event sequences. As a result, the corresponding pushdown automata typically have a relatively limited number of states, and the *shortest* path to an accepting state often requires no more than 10 transitions. So in this work, we uniformly set Q to 10. Although the precise distance can enhance the accuracy, it also increases computation overhead significantly, and a marginal improvement in accuracy may not justify the cost.

3.4 Input Prioritization

The seed can be divided into parts with different part sizes, *e.g.*, one byte for an input string or four bytes for an input integer array. Users can also specify the partition size to more precisely control the granularity of the mutations, thus enabling input partitioning. For inputs to protocol-based programs, we rely on the protocol specification and the structure of the input data. The key idea of Prioritize is to select the seed's part that is most related to triggering the bugs specified by ψ for mutation. There are two folds: 1) the part I_i is more related to the branches controlling the statements of ψ's events; 2) the events that may be generated by mutating I_i can advance the state to be closer to ψ's accepting state.

We calculate each part's priority based on the entropy-based information flow calculation [21, 23], which relates each seed part with $\mathcal{P}$'s conditions. The information for prioritization is collected by the instrumentations added before fuzzing (Section 3.2). Specifically, based on the runtime monitor instrumentation, we perform static analysis on the program, construct an inter-procedural control flow graph, identify the event instrumentation statements of the property in the program, and condition statements on which

Algorithm 3: Context-Free Property Oriented Input Prioritization

```
Prioritize($\mathcal{P}_I, s, \psi, M$)
Data: $\mathcal{P}_I$ is the instrumented program, $s$ is the selected input seed with $N$
      parts, $\psi$ is the bug property, and M is the set of control conditions
begin
    for $i \in \{1, ..., N\}$ do
        for $j \in \{1, ..., K\}$ do
            $m \leftarrow \mathsf{Mutate}_b(s, i)$
            $(V_b, V_s) \leftarrow \mathsf{Execute}_i(\mathcal{P}_I, m)$
            for $k \in \{1, ..., M\}$ do
                $(s, E, D_s) \leftarrow V_s[k]$    ▷ $D_s = D_s(\psi, s)$
                ▷ $D_s(\psi, \delta(s, e))$ is the distance to the accepting states
                  after executing event $e$ at state $s$
                $D_{min} \leftarrow \min_{e \in E \cap enable(\psi,s)} D_s(\psi, \delta(s, e))$
                if $D_{min} > D_s(\psi, s)$ then
                    $V[k] \leftarrow 0$    ▷ Ignore the kth branch.
                end
                else
                    $V[k] \leftarrow V_b[k] + \frac{1}{D_{min}+1}$
                    ▷ Consider both the distance on the monitor and the
                      distance that satisfies the conditions
                end
            end
            $\mathcal{MB}_i[j] \leftarrow (m, V)$
        end
        for $k \in \{1, ..., M\}$ do
            $\mathcal{MP}[i][k]$←IFS($\mathcal{MB}_i, k$) ▷ The ith part and the kth branch
        end
        $\mathcal{MP}[i] \leftarrow \max_{0 \le j \le M} \mathcal{MP}[i][j]$
    end
    return $\mathcal{MP}$
end
```

the event instrumentation statements have control dependence. Then, we add instrumentations before every branch statement b controlling the statements that can generate any event of ψ, in order to collect the information for entropy-based input prioritization, *e.g.*, the branch statement at Line 15 in Figure 2's example program.

Algorithm 3 shows the details of prioritizing the different seed parts. The inputs are the program after instrumentation, a seed input, and the bug's temporal property. The algorithm returns the prioritization of the seed's each part.

For each part of the seed, we mutate it and run $\mathcal{P}_I$ with the mutants K times (Line 2 - 5). Suppose M branch conditions have been instrumented with respect to ψ. Each running of $\mathcal{P}_I$ returns two M-dimensional vectors: V_b, containing condition fitness values, and V_s, where each element (s, E, D_s) denotes the current state, the set of possible events, and their distance information of the corresponding branch condition.

If branch statement b's condition is C, the instrumentation before b collect the following information.

- The distance to satisfy C. We employ the existing method of fitness functions [16] to calculate the distance. Until now, we have only considered the conditions of numeric variables. For example, if C is $a == b$ of two integer variables a and b, the fitness function is $a - b$. We use $V_b[k]$ to represent the distance value of the kth condition (Line 14).
- The current states of the runtime monitors and the events generated inside b. We use the state of the runtime monitor that was last updated to the current state. The events that can be generated

inside b are statically calculated. We use (s, E, D_s) (denoted by $V_s[k]$) to represent the current state (s), the possible events (E), and their distance information (D_s) of the kth condition (Line 7).

For each part of the input, we combine the two types of information (Lines 7&9) as each condition's information. The fitness values reflect the input part's relation with the condition, and the second part reflects the possibility of reaching accepting states by executing the statements inside the condition's branch. We use the shortest distance to the accepting states of the states that the current state can be advanced to by the events controlled by the condition to evaluate the possibility, which is defined as follows, where $enable(\psi, s)$ represents the set of events that are enabled at the state s in ψ, $\delta(s, e)$ represents the target state of performing the event e at the state s, and $D_s(\psi, \delta(s, e))$ represents the shortest distance to ψ's accepting states after executing e at state s.

$$\min_{e \in E \cap enable(\psi, s)} D_s(\psi, \delta(s, e)) \tag{1}$$

We disregard the branches whose guarded events increase the distance to the accepting states (Line 11); Otherwise, we combine these two values (Line 14) to one vector V as the information of the branch conditions. We record the information generated by executing each mutant (Line 18).

Then, we calculate each seed part's relation to each branch by an entropy-based method [21, 23] (Lines 20-22), and the calculation of *IFS* follows the standard process of calculating conditional information entropy [23], where $\mathcal{MB}_i$ is a K-sized vector $((m_1, V_1), ..., (m_K, V_K))$. In principle, if $IFS(\mathcal{MB}_i, k)$ is larger, the input's ith part is more related to the kth branch condition. We use the most related condition's value as each input part's priority (Line 23).

The parameter K controls the number of mutants generated. In principle, a larger K will improve the accuracy of the priority results. However, it will also increase the overhead of input prioritization. Thus, there is a balance between the overhead and the efficiency improvement.

3.5 Seed Energy

A seed's energy determines its mutation times and directly influences fuzzing's results. Unlike the energy strategy in coverage-oriented fuzzing, we propose a seed energy strategy with respect to the property ψ. For each seed s, we can get its transition times $T(s)$ and the shortest distance $D(s)$ to the accepting states. Then, we define $\mathsf{Energy}(s)$ as follows, where C is the seed unit number and S_p is the set of the seeds in the seed pool.

$$C \times \left(\frac{T(s)}{\max_{c \in S_p} T(c)} + \frac{1 + \min_{c \in S_p} D(c)}{D(s) + 1}\right) \tag{2}$$

In principle, if a seed can cause more state transitions and reach the closer states to ψ's accepting states, it is more likely to trigger the bug and should be mutated more times.

3.6 Discussion

We utilize two kinds of information to prioritize the different parts of the input with respect to the bug property ψ. The fitness values of the conditions are natural. The event information E in (s, E, D_s) is not accurate. Limited by static analysis's imprecision, we calculate E by statically matching the statements and ignoring the runtime object information. Besides, the entropy-based calculation method is also statistical and provides a lightweight evaluation of prioritization for fuzzing. More accurate prioritization can be obtained by heavyweight approaches, such as taint analysis [32] and symbolic execution [8], but it also introduces significant overhead.

4 Implementation and Evaluation

4.1 Implementation

We implemented our approach for C/C++ programs using CCMOP [39] as the runtime verification platform. Input prioritization instrumentation is built on LLVM [19]. Our fuzzing optimizations are integrated into two fuzzers (*i.e.*, AFL [40] and AFLNet [31]) for fuzzing C/C++ application programs and network protocol programs. Based on AFL and AFLNet, we have developed a new fuzzing guidance strategy that implements the mutant preservation in Algorithm 1 and the input prioritization in Algorithm 3.

4.2 Research Questions

To evaluate our method, we carried out experiments to answer the following research questions:

- **RQ1**: *effectiveness*, how effective is our method for fuzzing the program paths that violate the context-free property?
- **RQ2**: *efficiency*, how efficient is our method? Efficiency means finding the program paths that satisfy the context-free property in a shorter time.
- **RQ3**: *usefulness*, how useful is our method in revealing context-free bugs in real-world programs? Have we discovered any real unknown bugs?
- **RQ4**: *combination*, unlike the fuzzing guided by coverage, our method uses context-free property for guidance. How useful would it be if we combined the two guidance methods?

4.3 Experimental Setup

C/C++ benchmarks. Table 1 lists the C/C++ programs in our benchmark. All the programs are open-source real-world C/C++ programs. These programs are popular Github projects with high stars. The scales of these programs are also well-distributed, *i.e.*, ranging from 9k to 406k. We use 12 versions of these 7 programs, abstracting 17 context-free properties for fuzzing. It is worth noting that these properties exceed the expressive power of both regular grammars (finite-state automata) and ω-regular grammars (Büchi automata). Therefore, they must be represented using context-free grammars. *Due to the space limit, we give the details of the context-free properties in the supplementary document*[1]. Many of these programs have been used in related literature [24, 34].

The extraction of CFG properties. CFG properties for known bugs are manually extracted from CVE reports and GitHub issues. While these CVEs may initially appear to be related to memory errors, this could lead to a misunderstanding. However, through careful analysis of the programs and CVEs, we identified that the root cause stems from a mismatch in counting certain events. The resulting memory errors are merely one of the possible consequences

[1]Available at the artifact repo https://github.com/zbchen/CFPOFuzz .

Table 1: The benchmark C/C++ programs

Project	Language	Protocol	SLOC	Github Stars	Brief Description
Mujs	C	–	18.4k	791	A lightweight embedded JavaScript engine
lua	C	–	22.3k	8.9k	A lightweight compiler for scripting languages
Exiv2	C++	–	406.7k	913	A library and command-line tool for image
Luna	C	–	9.1k	955	An Amaranth HDL library for USB
TinyDTLS	C	DTLS	63.2k	105	An implementation of the DTLS protocol
Live555	C++	RTSP	53.8k	744	A library for streaming multimedia
OpenSSL	C	TLS	321k	25.4k	An implementations for the TLS protocols

of such mismatches. The properties we extracted, which are detailed in the appendix, involve relative event counts (e.g., equality, greater than, less than) that go beyond the expressive power of regular grammars such as LTL or ERE.

For unknown bugs, properties are partially derived from protocol RFCs (e.g., LV3, LV4, TD4) and partially from program documentation after analysis (e.g., LN1, LN2, MJ5, LN3). The extraction process involves: (1) identifying textual descriptions that reveal bug-triggering inputs or expected behaviors; (2) defining event-to-behavior mappings (e.g., abstracting pointer dereference as event Dere); and (3) specifying acceptable or forbidden event sequences using context-free grammars. All properties in the benchmark are independently validated by multiple authors against program semantics or official documentation.

Experimental configurations. All the context-free properties evaluated in this paper cannot be expressed in regular languages and Linear Temporal Logic. As we know, we are the first fuzzing approach for general non-regular context-free properties. Hence, we do not compare it with the existing tools that support only regular or Linear Temporal Logic properties. We use coverage-oriented fuzzing (denoted by **COV**) as the baseline, with AFL applied to application software, and AFLNet used for protocol software. **Please note that we use the monitor-instrumented version of the program for coverage-oriented fuzzing.** Besides, to evaluate the influence of our two novel algorithms, *i.e.*, the mutant preservation and the input prioritization, we have the following configurations:

- $\mathbf{CFPO_{cov}}$**+IP**, *i.e.*, our framework with both algorithms enabled.
- $\mathbf{CFPO_{cov}}$, *i.e.*, the variant by disabling the input prioritization.
- **CFPO**, *i.e.*, the variant that disables the input prioritization and only considers distance and the number of transitions in the mutant preservation (*i.e.*, disabling Lines 12-14 in Algorithm 1).
- **CFPO+IP**, *i.e.*, the variant that combines the input prioritization and the mutant preservation considering only distance and the number of transitions.
- **COV+IP**, *i.e.*, the variant that uses the input prioritization and the coverage-oriented guidance.

Hence, under six configurations, we run each fuzzing task, *i.e.*, a program and context-free property. The time threshold of fuzzing is set to 24 hours. Besides, the number K (Algorithm 3) of mutants generated for calculating input priorities is set to 10, since our experience and preliminary experimental validation indicate that $K = 10$ achieves the best balance between effectiveness and overhead. The unit number C of seed energy (Section 3.5) is 1000. All the experiments were carried out in parallel on a server with 128 3.1GHz cores and 256G memory. The operating system is Ubuntu 22.04. We ran the experiments five times to tackle the experimental errors and use the averaged values.

Table 2: Main Results (#T: the time (mins) for triggering the bug, #R: the ratio of the time spent on COV to the time spent on CFPOFuzz, and T/O means timeout) . $\hat{A}_{12}$: statistically significant values, #p: Mann-Whitney U test p values.

Program	Version	Prop	Bug ID	$\mathrm{CFPO_{cov}}$+IP	COV			
				#T	**#T**	**#R**	$\hat{A}_{12}$	**#p**
Mujs	1.0.6	*MJ1*	CVE-2019-11413	2.57	22.54	8.77	1.0	0.01
Mujs	1.0.6	*MJ2*	CVE-2019-12798	13.90	945.66	68.03	1.0	0.01
Mujs	1.0.8	*MJ3*	mujs-bug1	20.77	353.64	17.03	0.92	0.03
Mujs	1.0.9	*MJ4*	CVE-2020-24343	2.52	340.14	134.98	1.0	0.01
OpenSSL	1.0.2	*OS1*	CVE-2015-0291	201.99	1193.12	5.91	0.96	0.02
OpenSSL	1.1.0	*OS2*	CVE-2017-3730	17.82	966.98	54.26	1.0	0.01
OpenSSL	1.1.1	*OS3*	CVE-2021-3449	16.51	76.39	4.63	1.0	0.01
Exiv2	0.27.6	*EV1*	CVE-2023-44398	1065.88	T/O	1.35	0.9	0.03
lua	5.4.3	*LA1*	CVE-2021-43519	41.07	53.76	1.31	0.76	0.21
lua	5.4.2	*LA2*	CVE-2020-24370	43.75	69.12	1.58	0.96	0.02
Found bugs in total			-	**10**	**9**			
Average time usage (mins)			-	**142.68**	**546.13**	**3.83x**		

4.4 Experimental Results

4.4.1 *Results of RQ1 and RQ2.*

We use the existing CVEs or bugs to evaluate the effectiveness and efficiency. Table 2 presents the results. The first four columns list the program name, version, corresponding CFG property, and associated CVE or bug ID. The fifth column reports the time (**#T**) required by CFPOFuzz to generate the first input that triggers the context-free property violation. The next four columns present the time (**#T**) required by coverage-guided fuzzing to expose the same bug, the time ratio (**#R**) between coverage-based fuzzing and CFPOFuzz, and the statistical significance metrics: $\hat{A}_{12}$ and **#p**.

As shown by Table 2, for the 10 tasks in total, $\mathbf{CFPO_{cov} + IP}$ (CFPOFuzz) can generate the target inputs, *i.e.*, the ones triggering the bugs, for all tasks. Coverage-oriented method **COV** can find the target inputs for 9 tasks but fails to trigger a bug, *i.e.*, CVE-2023-44398, in 24 hours. For the program on which CFPOFuzz can find the bug but the baseline cannot, this is due to the large search space and the specific sequence of events defined by the property, which is challenging to satisfy. Although coverage-guided fuzzing achieve high coverage, it might still fail to reach the desired accepting state. In contrast, our CFPOFuzz, through effective exploration that progressively approach the accepting state, is able to satisfy the requirements of event execution sequence in a relatively short time. On average, $\mathbf{CFPO_{cov} + IP}$ takes 142.68 mins to detect the known bugs in the programs. However, **COV** takes an average of 546.13 mins to detect 9 out of 10 errors. In terms of the $\hat{A}_{12}$ statistic, $\mathbf{CFPO_{cov} + IP}$ performs significantly better than **COV** in most cases. The significance level of all programs is below 0.05, further indicating the significance of the test results.

To evaluate Algorithms 1 and 3 further, we conducted ablation experiments under different configurations, whose results are shown in Table 3. As indicated by Table 2 and 3, each algorithm can contribute to improving fuzzing's performance. The context-free property-oriented mutant preservation (*i.e.*, Algorithm 1) is more important than the input prioritization (*i.e.*, Algorithm 3). Besides, by comparing **CFPO** with $\mathbf{CFPO_{cov}}$ and **CFPO + IP** with $\mathbf{CFPO_{cov} + IP}$, we can conclude that considering coverage in Algorithm 1 (Lines 12-14) can improve the performance for most programs, indicating the necessity of keeping the mutants that are both promising in satisfying the property and improving coverage.

Table 3: Ablation Experimental Results (#T: the time (mins) for triggering the bug, #R: the ratio of the time spent on the configuration to the time spent on $\text{CFPO}_{\textbf{cov}}$+IP, and T/O means timeout), $\hat{A}_{12}$: statistically significant values.

Bug ID	COV+IP			CFPO			CFPO_{cov}			CFPO+IP		
	#T	#R	$\hat{A}_{12}$	#T	#R	$\hat{A}_{12}$	#T	#R	$\hat{A}_{12}$	#T	#R	$\hat{A}_{12}$
CVE-2019-11413	20.65	8.04	1.0	8.60	3.35	1.0	7.92	3.08	1.0	2.61	1.02	0.36
CVE-2019-12798	428.03	30.79	1.0	191.96	13.81	1.0	240.52	17.30	1.0	15.31	1.10	0.76
mujs-bug1	66.16	3.19	0.92	27.39	1.32	0.72	26.63	1.28	0.72	22.21	1.07	0.6
CVE-2020-24343	321.86	127.72	0.82	10.71	4.25	1.0	9.91	3.93	1.0	2.80	1.11	0.68
CVE-2015-0291	989.91	4.90	0.92	625.81	3.10	0.68	398.29	1.97	0.64	231.79	1.15	0.56
CVE-2017-3730	38.48	2.16	1.0	24.38	1.37	0.76	20.70	1.16	0.72	18.80	1.05	0.4
CVE-2021-3449	46.32	2.81	1.0	34.35	2.08	0.88	32.65	1.98	0.88	19.15	1.16	0.76
CVE-2023-44398	T/O	1.35	0.9	T/O	1.35	0.9	T/O	1.35	0.9	1313.29	1.23	0.76
CVE-2021-43519	48.01	1.17	0.8	47.59	1.16	0.84	46.55	1.13	0.84	45.16	1.10	0.76
CVE-2020-24370	57.44	1.31	0.96	53.68	1.23	0.88	52.22	1.19	0.88	51.11	1.17	0.76
Found bugs in total	**9**			**9**			**9**			**10**		
Average time usage (mins)	**345.69**	**2.42x**		**246.45**	**1.73x**		**227.54**	**1.59x**		**172.22**	**1.21x**	

Answer to RQ1: *Our method can effectively detect all the known context-free bugs in the benchmark programs, with an average time cost below 142.68 mins. Besides, both algorithm 1 and algorithm 3 benefit to our method.*

For the efficiency, as indicated by the **#R** column in Table 2, $\textbf{CFPO}_{\textbf{cov}}$**+IP** has achieved great speedups in triggering the bugs compared with **COV**, and the average speedup is 3.83x. Besides, as shown in Table 3, compared with **COV+IP**, **CFPO**, $\textbf{CFPO}_{\textbf{cov}}$, and **CFPO+IP**, $\textbf{CFPO}_{\textbf{cov}}$**+IP** can achieve speedups for most programs. The average speedups are 2.42x, 1.73x, 1.59x, and 1.21x respectively, indicating each algorithm's necessity.

To reduce randomness, we conducted statistical analysis on the results. Considering the time cost of CFPOFuzz ($\textbf{CFPO}_{\textbf{cov}}$**+IP**) and other fuzzers, Vargha-Delaney $\hat{A}_{12}$ values measures the probability of CFPOFuzz performing better than other fuzzers (the higher, the better), and Mann-Whitney U p values measures the statistical significance of performance gains (the lower, the better). As indicated by the table, in most cases, CFPOFuzz performs consistently and significantly better than the others.

Answer to RQ2: *Compared with* **COV***,* **COV+IP***,* **CFPO***,* $\textbf{CFPO}_{\textbf{cov}}$*, and* **CFPO + IP***, CFPOFuzz on average achieves 3.83x, 2.42x, 1.73x, 1.59x, and 1.21x speedups for finding the first target input, respectively.*

4.4.2 Results of RQ3. To evaluate the usefulness of CFPOFuzz for detecting bugs in real-world programs, we extract context-free properties from program functionality, implementation details, and protocol RFC specifications.

We discovered seven new zero-day bugs, as shown in Table 4. The first column indicates the property, the second shows the program and version, the third provides a description of the properties. These bugs will cause the program to crash or specification violations. The error types of these bugs encompass null pointer dereferencing(LN1), floating point exceptions(LN2), out-of-bounds access(LN3), infinite loops(MJ5), and denial of service(LV3,LV4,TD4). We have reported these bugs during the process of verification and remediation. For the 7 zero-day bugs, 3 of them can be detected by AFL or AFLNet (on the monitor instrumented version of the programs) in 24 hours. There are 4 bugs (MJ5, LV3, LV4, and TD4) that AFL or AFLNet cannot find in 24 hours. These results demonstrate the usefulness of CFPOFuzz.

Table 4: Detected Unknown Zero-day Bugs.

Prop	Program	Description of the context-free property
LN1	`luna`(0.1.1)	$S \to A\,S\,B \mid B\,S\,A \mid S\,S \mid \epsilon$ **(fail)** The number of calls to the **A**(`scan_string()`) function is **not equal to** the number of **B**(`buf_assignment`) operations.
LN2	`luna`(0.1.1)	$S \to Q\,B \mid S\,B \mid S\,S$ $Q \to A\,Q\,B \mid B\,Q\,A \mid Q\,Q \mid \epsilon$ **(match)** The number of times **A**(`Selfexpr_notnull`) is **fewer than** the number of calls to the **B**(`visit_unary_op()`) function.
LN3	`luna`(0.1.1)	$S \to Q\,P \mid S\,P \mid S\,S$ $P \to B \mid C$ $Q \to A\,Q\,P \mid P\,Q\,A \mid Q\,Q \mid \epsilon$ **(match)** The number of occurrences of the **A**(`RK_Cnot0`) event is **fewer than** the combined number of occurrences of the **B**(`LUNA_OP_MOD`) and **C**(`LUNA_OP_DIV`) events.
MJ5	`mujs`(1.0.9)	$S \to A\,S\,B \mid B\,S\,A \mid S\,S \mid \epsilon$ **(fail)** The number of calls to the **A**(`jsR_run()`) function is **not equal to** the number of occurrences of the **B**(`OP_RETURN`) event.
LV3	`Live555`(0.92)	$S \to A\,Q \mid A\,S \mid S\,S$ $Q \to A\,Q\,B \mid B\,Q\,A \mid Q\,Q \mid \epsilon$ **(match)** After establishing the connection, the number of times the first valid **A**(`setup request`) is received is **greater than** the number of times a **B**(`valid MediaSource`) is created.
LV4	`Live555`(0.92)	$S \to Q\,B \mid S\,B \mid S\,S$ $Q \to A\,Q\,B \mid B\,Q\,A \mid Q\,Q \mid \epsilon$ **(match)** After establishing the connection, the number of **A**(`valid MediaTable entries`) is **fewer than** the number of **B**(`valid setup requests`).
TD4	`TinyDTLS`(0.9-rcl)	$S \to Q\,P \mid S\,P \mid S\,S$ $P \to B\,C$ $Q \to A\,Q\,P \mid P\,Q\,A \mid Q\,Q \mid \epsilon$ **(match)** The number of times the server rejects and **B**(`sends an Alert`) is **fewer than** the number of occurrences of the sequence where the server receives a **B**(`ClientHello`), gives a **B**(`HelloVerifyRequest`) `response`, and then receives an `over-large packet`.

Answer to RQ3: *Our approach detects seven new zero-day bugs abstracted as different context-free properties.*

4.4.3 Results of RQ4. To explore the impact of combining our method with the most popular coverage-guided fuzzing approach (**COV**), we also implemented the configurations of **COV+**$\textbf{CFPO}_{\textbf{cov}}$ and **COV+**$\textbf{CFPO}_{\textbf{cov}}$**+IP** in our experiments. It should be noted that the coverage of COV is different from that of $\textbf{CFPO}_{\textbf{cov}}$. In the $\textbf{CFPO}_{\textbf{cov}}$ configuration, the growth of coverage is considered only when both the distance to the accepting state and the number of executions of events are equal to D_m and T_m (Algorithm 1 Line 12), and only a very small subset of COV is taken into account. In the COV configuration, the coverage has the same priority as the context-free property-oriented mutant preservation, *i.e.*, a mutant will be kept if it improves coverage, *or* increases the number of

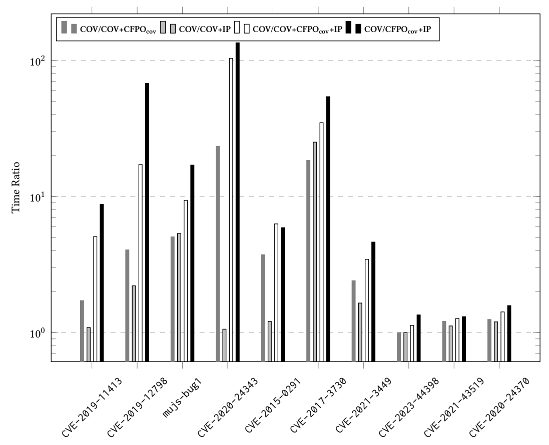

Figure 6: Time ratio of coverage and other methods.

transitions, *or* decreases the distance to the accepting states. Figure 6 shows the results. The X-axis shows the task IDs of the existing bugs, and the Y-axis shows the ratio of the time **COV** takes to find the first target input to the time taken by the different configurations. A ratio greater than 1 indicates that the configuration finds the target bug more efficiently than **COV**; Otherwise, it is less efficient than **COV**. For the same task, a larger value on the vertical axis indicates higher efficiency in discovering the first target input.

As shown by the figure, **COV+CFPO$_{\mathbf{cov}}$**, **COV+IP**, and **COV+CFPO$_{\mathbf{cov}}$+IP** outperform **COV** in 9, 9, and 10 tasks, respectively, with average speedups of 2.40x, 1.58x, and 3.21x compared to **COV**. Hence, both algorithms can improve the coverage-oriented method's efficiency. Besides, it can be observed that **CFPO$_{\mathbf{cov}}$+IP** outperforms **COV+CFPO$_{\mathbf{cov}}$**, **COV+IP**, and **COV+CFPO$_{\mathbf{cov}}$+IP** in 10, 10, and 9 tasks, respectively, with the average speedups of 1.59x, 2.42x, and 1.19x. These results demonstrate that the coverage-oriented method may influence our algorithms' performance if we have the same priority on coverage for mutant preservation.

Answer to RQ4: *The combinations of* COV *with* CFPO$_{\text{cov}}$, IP*, and* CFPO$_{\text{cov}}$+IP *achieve an average improvement of **2.40x**, **1.58x**, and **3.83x**, respectively. However, the combinations are still less effective than* CFPO$_{\text{cov}}$+IP.

4.5 Threats to Validity

The main threats are external. The lack of existing work supporting fuzzing or verification of context-free properties (especially for C/C++ programs) complicates experimental comparison. Internal threats arise from two factors. First, the method's randomness, which we mitigated by running five experimental repetitions to reduce errors. Second, the method's overhead, including distance calculation and prediction, is influenced by the complexity of the context-free property and the length of the event sequence. Additionally, the overhead introduced by instrumentation depends on event frequency. If events are sparse, overhead is minimal; otherwise, it may be significant, particularly for memory operation events. In our experiments, CFPOFuzz's overhead for these two factors remains acceptable (1% on average) over 24 hours of fuzzing.

5 Related Work

Directed fuzzing. Directed grey-box fuzzing [17, 28–30, 37, 38, 40] is a very active topic in fuzzing for improving fuzzing's efficiency. The existing approaches propose different guiding methods with respect to different fuzzing targets, such as improving coverage [17, 30, 40], memory-related bug detection [38], and typestate bug detection [37], to name a few. Our framework is also a directed grey-box fuzzing framework that targets context-free properties. UAFL [37] and LTL-Fuzzer [24] are the most related work and inspire our work. However, as discussed in Section 1, both UAFL and LTL-Fuzzer have inherent limitations. First, *our framework supports properties with expressiveness beyond that of FSMs and Büchi automata*, exceeding the expressiveness of the specifications they can handle. Besides, UAFL may encounter false alarms and miss bugs. It is also the reason why UAFL concentrates on fuzzing use-after-free bugs. Compared with UAFL, our method does not have false alarms, and we can detect all the bugs possibly triggered by the same-sized input. LTL-Fuzzer provides a prefix controller but no other guidance for seed mutation. Unlike LTL-Fuzzer, our method considers state transition information in input prioritization.

Runtime verification. Runtime verification [18] is also related to our fuzzing framework. The runtime monitoring in our framework is based on CCMOP[39], which is a runtime verification framework for C/C++ programs. Different approaches of runtime verification differ in the following aspects: the supported properties, such as LTL [6], FSM [3] and hyper-properties [7, 11], to name a few; the programs of different languages, including Java [3, 14], C/C++ [9, 10], *etc*; the instrumentation methods, such as aspect-based method [14] and IR-based instrumentation [36]. It is interesting to leverage more advanced runtime verification techniques to improve the framework's ability and efficiency.

6 Conclusion and Future work

Fuzzing temporal bugs is challenging. This paper proposes a general context-free property-oriented fuzzing framework. We propose two algorithms for seed preservation and input prioritization with respect to a context-free property inside our framework. We have implemented CFPOFuzz for C/C++ programs based on state-of-the-art fuzzing platforms. In contrast to coverage-guided fuzzing, our method achieves 3.83x speedups in generating the first input that exposes the bug, while also identifying 7 previously unknown zero-day bugs. The results of the extensive experiments indicate the effectiveness and efficiency of CFPOFuzz. The future work lies in several aspects: 1) more extensive evaluation on more real-world programs; 2) more applications in some specific backgrounds.

Data-Availability Statement

Our artifact is available at the following URL: https://github.com/zbchen/CFPOFuzz .

Acknowledgments

This research was supported by National Key R&D Program of China (No. 2024YFF0908003) and the NSFC Program (No. 62172429, 62032024, and 62372162). We also thank the anonymous reviewers for their valuable feedback.

References

[1] Abhishek Aarya, Oliver Chang, Max Moroz, Martin Barbella, and Jonathan Metzman. 2019. Open sourcing ClusterFuzz. *Google, Inc. Feb* (2019).

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison-Wesley Longman Publishing Co., Inc., USA.

[3] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. 2005. Adding Trace Matching with Free Variables to AspectJ. *SIGPLAN Not.* 40, 10 (2005), 345–364.

[4] Remzi H. Arpaci-Dusseau. 2017. Operating Systems: Three Easy Pieces. *login Usenix Mag.* 42, 1 (2017).

[5] Shuangpeng Bai, Zhechang Zhang, and Hong Hu. 2024. CountDown: Refcount-guided Fuzzing for Exposing Temporal Memory Errors in Linux Kernel. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security* (Salt Lake City, UT, USA) *(CCS '24)*. Association for Computing Machinery, New York, NY, USA, 1315–1329.

[6] Andreas Bauer, Martin Leucker, and Christian Schallhart. 2011. Runtime Verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.* 20, 4 (2011), 64 pages.

[7] Borzoo Bonakdarpour and Bernd Finkbeiner. 2016. Runtime Verification for HyperLTL. In *Runtime Verification* (Madrid, Spain) *(Lecture Notes in Computer Science)*. Springer International Publishing, Cham, 41–45.

[8] Cristian Cadar and Koushik Sen. 2013. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM* 56, 2 (2013), 82–90.

[9] Zhe Chen, Zhemin Wang, Yunlong Zhu, Hongwei Xi, and Zhibin Yang. 2016. Parametric Runtime Verification of C Programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2016)*. Springer Berlin Heidelberg, Berlin, Heidelberg, 299–315.

[10] Zhe Chen, Junqi Yan, Shuanglong Kan, Ju Qian, and Jingling Xue. 2019. Detecting Memory Errors at Runtime with Source-Level Instrumentation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. ACM, New York, NY, USA, 341–351.

[11] Bernd Finkbeiner, Christopher Hahn, Marvin Stenger, and Leander Tentrup. 2018. RVHyper: A Runtime Verification Tool for Temporal Hyperproperties. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer International Publishing, Cham, 194–200.

[12] JE Hopcroft. 2001. Introduction to Automata Theory, Languages, and Computation.

[13] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. 2007. *Introduction to Automata Theory, Languages, and Computation, 3rd Edition.* Addison-Wesley.

[14] Dongyun Jin, Patrick O'Neil Meredith, Choonghwan Lee, and Grigore Roşu. 2012. JavaMOP: Efficient Parametric Runtime Monitoring Framework. In *Proceedings of the 34th International Conference on Software Engineering* (Zurich, Switzerland) *(ICSE '12)*. IEEE Press, 1427–1430.

[15] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. 2001. An Overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP '01, Vol. 2072)*. Springer-Verlag, Berlin, Heidelberg, 327–353.

[16] Gwangmu Lee, Woochul Shim, and Byoungyoung Lee. 2021. Constraint-guided Directed Greybox Fuzzing. In *30th USENIX Security Symposium*. USENIX Association.

[17] Caroline Lemieux and Koushik Sen. 2018. FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE 2018)*. Association for Computing Machinery, New York, NY, USA, 475–485.

[18] Martin Leucker and Christian Schallhart. 2009. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming* 78, 5 (2009), 293–303. The 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'07).

[19] LLVM. 2025. The LLVM Compiler Infrastructure. https://llvm.org.

[20] Qingzhou Luo, Yi Zhang, Choonghwan Lee, Dongyun Jin, Patrick O'Neil Meredith, Traian Florin ŞerbănuŢă, and Grigore Roşu. 2014. RV-Monitor: Efficient Parametric Runtime Verification with Simultaneous Properties. In *Runtime Verification*, Vol. 8734. Springer, Cham, 285–300.

[21] David J. C. MacKay. 2003. *Information theory, inference, and learning algorithms.* Cambridge University Press.

[22] Zohar Manna and Amir Pnueli. 1992. *The Temporal Logic of Reactive and Concurrent Systems - Specification.* Springer.

[23] Wes Masri and Andy Podgurski. 2009. Measuring the Strength of Information Flows in Programs. *ACM Trans. Softw. Eng. Methodol.* 19, 2, Article 5 (2009), 33 pages.

[24] Ruijie Meng, Zhen Dong, Jialin Li, Ivan Beschastnikh, and Abhik Roychoudhury. 2022. Linear-time Temporal Logic guided Greybox Fuzzing. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. 1343–1355.

[25] Bertrand Meyer. 1992. *Eiffel: The Language.* Prentice-Hall, Inc., USA.

[26] Barton P. Miller, Louis Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* 33, 12 (1990), 32–44.

[27] Glenford J. Myers. 2004. *The art of software testing (2. ed.).* Wiley.

[28] Manh-Dung Nguyen, Sébastien Bardin, Richard Bonichon, Roland Groz, and Matthieu Lemerre. 2020. Binary-level Directed Fuzzing for Use-After-Free Vulnerabilities. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. USENIX Association, San Sebastian, 47–62.

[29] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2020. ParmeSan: Sanitizer-guided Greybox Fuzzing. In *29th USENIX Security Symposium*. USENIX Association, 2289–2306.

[30] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic Fuzzing with Zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. ACM, New York, NY, USA, 329–340.

[31] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2020. AFLNET: A Greybox Fuzzer for Network Protocols. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. 460–465.

[32] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*. IEEE Computer Society, 317–331.

[33] Kostya Serebryany. 2015. libFuzzer - a library for coverage-guided fuzz testing. *LLVM project* (2015).

[34] Abhishek Shah, Dongdong She, Samanway Sadhu, Krish Singal, Peter Coffman, and Suman Jana. 2022. MC2: Rigorous and Efficient Directed Greybox Fuzzing. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) *(CCS '22)*. Association for Computing Machinery, New York, NY, USA, 2595–2609.

[35] Robert E. Strom and Shaula Yemini. 1986. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering* SE-12, 1 (1986), 157–171.

[36] Dwight Guth Traian Florin Şerbănuţă. 2020. LLVMMop. https://github.com/runtimeverification/llvmmop/.

[37] Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, and Yulei Sui. 2020. Typestate-Guided Fuzzer for Discovering Use-after-Free Vulnerabilities. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. ACM, New York, NY, USA, 999–1010.

[38] Cheng Wen, Haijun Wang, Yuekang Li, Shengchao Qin, Yang Liu, Zhiwu Xu, Hongxu Chen, Xiaofei Xie, Geguang Pu, and Ting Liu. 2020. MemLock: Memory Usage Guided Fuzzing. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. ACM, New York, NY, USA, 765–777.

[39] Yongchao Xing, Zhenbang Chen, Shibo Xu, and Yufeng Zhang. 2023. CCMOP: A Runtime Verification Tool for C/C++ Programs. In *Runtime Verification: 23rd International Conference, RV 2023, Thessaloniki, Greece, October 3–6, 2023, Proceedings* (Thessaloniki, Greece). Springer-Verlag, Berlin, Heidelberg, 339–350.

[40] Michal Zalewski. 2025. American Fuzzy Lop. https://lcamtuf.coredump.cx/afl/.

[41] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2019. The Fuzzing Book. In *The Fuzzing Book*. Saarland University.