MUTATO: Enhancing Fuzz Drivers with Adaptive API Option Mutation

Shuangxiang Kan*, Xiao Cheng†, and Yuekang Li*
*University of New South Wales, Australia
shuangxiang.kan@unsw.edu.au, yuekang.li@unsw.edu.au
†Macquarie University, Australia
xiao.cheng@mq.edu.au

***Abstract*—Fuzz testing is a cornerstone technique for uncovering vulnerabilities and improving the reliability of software systems. Recent studies reveal that the primary bottleneck in modern coverage-guided fuzzing lies not within the fuzzers themselves, but in the construction of fuzz drivers—particularly their limited flexibility in exploring option parameters within library APIs. Existing approaches predominantly focus on mutating input data, often neglecting configuration options that fundamentally influence API behavior and may conceal critical vulnerabilities. To address this gap, we present MUTATO, a new multi-dimensional fuzz driver enhancement approach that systematically and adaptively mutates both input data and option parameters using a coverage-guided, epsilon-greedy strategy. Unlike prior work that requires intrusive modifications to fuzzers or targets only program-level options, MUTATO operates at the driver level, ensuring fuzzer-agnostic applicability and seamless integration with both manual and automatically generated drivers. We further introduce an option parameter fuzzing language (OPFL) to guide the enhancement of drivers. Extensive experiments on 10 widely used C/C++ libraries demonstrate that MUTATO-enhanced drivers achieve, on average, 14% and 13% higher code coverage compared to original AFL++ and LibFuzzer drivers, respectively, and uncover 12 previously unknown vulnerabilities, including 3 CVEs. Notably, we identified 4 vulnerabilities within 5 hours in APIs that OSS-Fuzz had failed to detect despite more than 18,060 hours of fuzzing effort.**

I. INTRODUCTION

Fuzz testing has emerged as a powerful technique for identifying vulnerabilities and improving the reliability of software systems. Recent research by Google [1] has demonstrated that a primary bottleneck in modern coverage-guided greybox fuzzing does not reside in the fuzzers themselves (e.g., AFL++ [2], LibFuzzer [3]), but rather in the design and implementation of fuzz drivers (or fuzz harnesses) [4], [5]. Their analysis reveals that many fuzz blockers—particularly those limiting code coverage—are input-independent and arise from deficiencies in driver construction, such as missing function calls or inadequate parameter configurations. This observation highlights the importance of *enhancing driver flexibility* as a promising direction for overcoming coverage plateaus.

In the context of fuzzing library APIs, a critical challenge lies in the systematic exploration of *option parameters* that fundamentally influence API behavior. Current fuzzing approaches predominantly focus on mutating input data while neglecting these configuration options. For example, when fuzzing the cJSON library, APIs like `cJSON_ParseWithOpts()` accept option flags that alter parsing behavior. A fuzz driver using fixed parsing options while only varying the JSON input string may fail to explore execution paths that depend on specific option configurations, potentially leaving vulnerabilities undiscovered. While modifying existing fuzzers to support option exploration is possible, it typically requires substantial changes for each fuzzer variant, resulting in increased development overhead and reduced generalizability. In contrast, enhancing fuzz drivers to systematically explore both input data and option parameters offers a more elegant, scalable, and fuzzer-agnostic approach that avoids the complexities inherent in modifying core fuzzing infrastructure.

Prior research on option exploration in fuzzing has predominantly concentrated on *program-level options*, such as command-line arguments and configuration files, rather than on *option parameters within library APIs*. Approaches like POWER [6] and ZigZagFuzz [7] incorporate option mutation directly into the fuzzer, while CarpetFuzz [8] and ProphetFuzz [9] utilize natural language processing and large language models to identify high-risk option combinations from documentation. ConfigFuzz [10] encodes program options within the fuzzable input to leverage existing mutation operators. However, these techniques are primarily designed for program-level configuration and do not address the unique challenges posed by option parameters in library APIs, such as their tight coupling with API semantics and the need for systematic exploration within the driver context. Concurrently, advances in automatic fuzz driver generation [11], [4], [12], [13] have improved the construction of fuzzing drivers, but these efforts typically emphasize basic driver structure and program input mutation, offering limited support for the systematic exploration of API option parameters.

A straightforward approach to enabling option parameter mutation is to embed API option values in the first few bytes of the input. However, this strategy conflates option parameters

Xiao Cheng is the corresponding author.

Network and Distributed System Security (NDSS) Symposium 2026
23 - 27 February 2026 , San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.230820
www.ndss-symposium.org

and main input data into *a single mutation domain*, which can significantly undermine fuzzing effectiveness. For instance, AFL and AFL++ employ the `auto_extra` feature, which automatically extracts recurring byte patterns—often from the input prefix—as dictionary tokens for subsequent mutation. When option values are embedded in the prefix, these tokens are inappropriately applied to mutating the main input data as they are only meaningful as option parameters. This misapplication results in wasted mutation efforts, as the semantics of option parameters and input data are fundamentally distinct. These observations underscore the necessity of treating option parameters and main input data as separate mutation domains, each requiring tailored mutation strategies.

To address this critical gap in effective API option parameters mutation, we introduce MUTATO, a novel *multi-dimensional* fuzzing methodology that advances library API fuzzing by enabling coordinated, dynamic mutation along two dimensions: option parameters and input data. MUTATO augments existing C/C++ API fuzz drivers with embedded option mutation logic, guided by an adaptive option parameter mutation strategy based on coverage-based feedback and the epsilon-greedy algorithm [14], [15], [16], [17], [18]. Specifically, our adaptive approach intelligently balances exploration and exploitation: when input mutations under the current option configuration continue to yield increased coverage, the driver maintains the existing options; upon encountering a coverage plateau (e.g., after 10k executions without new coverage), the driver mutates option parameters to probe alternative behaviors. This systematic approach enables the fuzz driver to explore both valid and invalid option values, thereby uncovering a broader spectrum of API behaviors.

This design confers three principal advantages. First, by implementing option mutation logic at the driver level, MUTATO remains **fuzzer-agnostic**, requiring no modifications to existing fuzzers and thus ensuring broad compatibility. Second, the methodology is **independent of specific fuzz driver generation techniques**, allowing seamless integration with both manually crafted and automatically generated drivers. Third, MUTATO enables **systematic and adaptive exploration of option parameters**, as opposed to indiscriminate random mutation, thereby ensuring that each option variation is meaningfully evaluated for its impact on API behavior. By triggering option mutations only upon reaching coverage plateaus and prioritizing those that facilitate the discovery of new execution paths, MUTATO avoids the inefficiency of random option changes while preserving the integrity of both the fuzzer architecture and the evolving input corpus. This principled approach directly addresses the limitations identified in prior work and establishes a rigorous foundation for comprehensive library API fuzzing.

Our work makes the following key contributions:

- We present MUTATO, a novel fuzz driver enhancement methodology that is both fuzzer-agnostic and independent of specific fuzz driver generation techniques, thereby enabling broad applicability across diverse fuzzing environments.
- We propose a coverage-guided, epsilon-greedy strategy for adaptive option parameter mutation, enabling efficient exploration based on coverage feedback.
- We design OPFL, a domain-specific language for specifying option parameter fuzzing, simplifying user guidance and integration.
- Our experimental results demonstrate that MUTATO-enhanced fuzz drivers achieve, on average, 14% and 13% higher code coverage compared to the original AFL++ and LibFuzzer drivers, respectively. Furthermore, MUTATO-enhanced drivers discover 12 previously unknown vulnerabilities, 3 of which have been assigned CVE identifiers. Notably, we identify 4 vulnerabilities within 5 hours in APIs that had remained undetected despite over 18,060 hours of continuous fuzzing by the OSS-Fuzz project. Ablation studies also validate that MUTATO's core design choices — adaptive option selection, exploration of valid and invalid options, epsilon value selection, and option change frequency — collectively enhance the effectiveness of option-aware fuzzing.

The source code of MUTATO is available at: doi:10.5281/zenodo.15909441.

II. Background and Problem Formulation

A. Fuzzing for Library APIs

Security vulnerabilities in library APIs pose significant risks due to the pervasive integration of libraries across diverse software systems. Traditional fuzzing approaches, which rely on fuzzing engines [2], [19], [20] alone, are often insufficient for libraries because APIs typically lack dedicated entry points and fixed input formats. To address this limitation, fuzz drivers (or fuzz harnesses) [4], [5], [11] are employed to create a controlled testing environment for library APIs. These drivers are responsible for generating inputs, invoking the target APIs, and processing their outputs, thereby acting as an interface between the fuzzing engine and the library under test. This design enables more systematic and effective exploration of API behaviors. A particular challenge arises when APIs accept option parameters—user-controllable configuration settings that influence the execution semantics of the API. Such parameters act as behavioral switches or modifiers, so that, for the same primary input, varying the option parameter values can induce divergent execution paths, output formats, or processing strategies within the API.

B. Problem Formulation

To systematically approach the problem of fuzzing option parameters, we first need to establish a formal framework that precisely defines the components involved. This formalization helps us reason about the problem space and develop effective strategies for exploring it.

Definition 1: Let L denote the library under test, and let $A = \{a_1, a_2, \ldots, a_n\}$ be the set of APIs in L that accept option parameters. For each API $a_i \in A$, we define:

- X_i: The set of all possible inputs to a_i, which may be either finite or countably infinite.

- O_i: The set of all possible assignments to the option parameters of a_i, where each element corresponds to a specific configuration of option values.
- $O_i^{\text{valid}} \subseteq O_i$: The subset of option parameter assignments that conform to the API specification and are considered valid. Valid values are determined based on (1) predefined option values explicitly documented in header files or library documentation, or (2) when no predefined values exist, the valid range of the parameter's data type.
- $O_i^{\text{invalid}} = O_i \setminus O_i^{\text{valid}}$: The subset of type-compatible option parameter assignments that violate semantic constraints (e.g., out-of-range integers, malformed strings, or undefined enumeration values) which may pass compilation but trigger error handling routines or induce undefined behavior during execution. Invalid values fall outside these API definitions and can be derived using standard boundary testing (e.g., values slightly outside the allowed range).

The option parameter values O_i are *user-controllable* configuration settings that influence or direct the execution behavior of an API, distinct from the primary input data X_i. These parameters serve as behavioral switches or modifiers such that, for the same input $x \in X_i$, different option parameter values $o_j, o_k \in O_i$ can induce divergent execution paths, output formats, or processing strategies within a_i.

Similar to program options used in [10], we categorize API option parameters into the following types:

- Boolean options with values $\{0, 1\}$ and potential invalid values $\{-1, 2, \ldots\}$.
- Enumerated choices with predefined valid values $E = \{e_1, e_2, \ldots, e_k\}$ where E is a finite enumerated set, and all other values considered invalid.
- Numeric options with valid ranges $[l, u]$ and invalid values outside this range.
- String options with predefined valid format patterns $S = \{s_1, s_2, \ldots, s_j\}$ where S is a set of valid string patterns, and invalid malformed strings that don't match these patterns.

Building on this characterization of option parameters, we formally model each API a_i as a function that maps a pair consisting of an input and an option assignment to an execution outcome and associated code coverage information:

$$a_i : X_i \times O_i \longrightarrow R_i \times \text{Cov}_i,$$

where R_i denotes the result space, encompassing all possible execution outcomes, including normal results and potential abnormal behaviors (such as crashes, memory corruption, or buffer overflows). The set Cov_i represents program execution coverage metrics related to a_i, which may include branch, path, basic block, or function coverage.

We formulate the option parameter fuzzing problem as follows:

Given a library API a_i, the objective is to systematically explore combinations of option values O_i and inputs X_i in order to (1) maximize code coverage $\left|\bigcup_{(x,o)\in X_i\times O_i} Cov_i(a_i(x,o))\right|$, and (2) discover as many instances as possible of abnormal behavior.

III. A Motivating Example

As discussed in [1], a well-designed fuzz driver can significantly improve fuzzing efficiency, and many works [11] have been devoted to automated driver generation to reduce manual costs. While current fuzzers excel at mutating raw input data (e.g., files, network packets), they largely treat API option arguments as *static or hardcoded values*. This approach overlooks the critical reality that many API bugs manifest only under specific combinations of inputs and options.

```
diff --git a/cjson_read_fuzzer.c b/cjson_read_fuzzer.c
index 12345678..abcdef01 100644
--- a/cjson_read_fuzzer.c
+++ b/cjson_read_fuzzer.c
@@ -1,3 +1,27 @@
+ // Tracks previous coverage measurement
+ unsigned int pre_cov = 0;

+ // initial default option
+ cJSON_bool cur_require_termination = 1;

+ int check_coverage_growth() {
+    // Interval-based coverage monitoring: computes
      ↪ current coverage metrics and compares against
      ↪ baseline (pre_cov) to detect gains
+    ...omited implementation...
+ }
+

+ cJSON_bool get_require_termination(int cov_growth,
+                                    int epsilon)
+ {
+    // Option parameter mutation strategy: combines
      ↪ coverage feedback with epsilon-greedy strategy
      ↪ to decide whether to keep using the current
      ↪ option or try a new one
+    ...omited implementation...
+ }
+
  int LLVMFuzzerTestOneInput(const uint8_t* data,
                     size_t size)
  {
@@ -9,8 +33,14 @@ int LLVMFuzzerTestOneInput(const
      ↪ uint8_t* data,
     cJSON *json;
     size_t offset = 4;
     int require_termination;

     if(size <= offset) return 0;
     if(data[size-1] != '\0') return 0;
     if(data[0] != '1' && data[0] != '0') return 0;

     require_termination = data[0] == '1' ? 1 : 0;

+     int cov_growth = check_coverage_growth();
+     new_require_termination =
+         get_require_termination(cov_growth, 95);
-     json = cJSON_ParseWithOpts((const char*)data,
-                         NULL, require_termination);
+     json = cJSON_ParseWithOpts((const char*)data,
+                     NULL, new_require_termination);
      if(json == NULL) return 0;

      cJSON_Delete(json);
  }
```

Listing 1: Comparison of the original and option-aware fuzz drivers for `cjson_read_fuzzer.c` [21].

Limitations of Existing Approaches. Both manually crafted and automatically generated fuzz drivers predominantly focus on mutating input data, often neglecting the influence of other arguments—particularly option parameters—on the behavior of target APIs. Only a minority of fuzz drivers attempt to vary option parameters during fuzzing, and even in such cases, the strategies employed are typically simple and limited in scope. For example, Listing 1 presents the `cjson_read_fuzzer.c` [21] driver, contributed by the `cJSON` [22] developers to the OSS-Fuzz project, where ℓ_{37} illustrates that the third argument, `require_termination`, is determined solely by the first byte of the input data. This approach exemplifies a common pattern among fuzz drivers that attempt to control option parameters via prepending option-control bytes in input data (notably, most fuzz drivers do not vary options at all during fuzzing). However, such simplistic mechanisms suffer from several inherent limitations:

- *Input Customization Overhead*: The `cJSON` driver requires customized inputs where option values are encoded in prefix bytes (e.g., `data[0..3]`) before the actual JSON string content like {`bf["Sunday",...,"Saturday"]`}. This approach not only increases manual effort but also lacks generalizability. For instance, it becomes impractical for file formats with strict header requirements like PDF or PNG where prepending option-control bytes would invalidate the format of the input, causing immediate rejection by the API due to format non-compliance and preventing further exploration of API behaviors. Furthermore, when embedding options within inputs, the fuzzing engine cannot distinguish which parts represent options versus actual input data, preventing systematic exploration of the option space during the fuzzing.
- *Stateless Option Selection*: Options are selected based solely on the current input, without considering historical execution context. When an input achieves higher coverage under a specific option value (e.g., `option=1`), the fuzzer should maintain this value while mutating inputs to thoroughly explore its impact. However, current approaches may arbitrarily switch to different options (e.g., `option=0`), disrupting the systematic exploration of API behaviors under promising option configurations.
- *Legal-Only Bias*: Fuzz drivers should test both valid and invalid option values to thoroughly evaluate the robustness of the API under test. Option parameters (e.g., file modes or boolean flags) usually have fixed, limited valid values, unlike non-option parameters (e.g., buffer lengths) with dynamic ranges, making option validation simpler but critical. However, most drivers only focus on legal options, missing potential vulnerabilities triggered by illegal values. For example, in C where boolean values are typically represented by integers, the `cJSON_ParseWithOpts` assume `require_termination` is always 0 or 1 (e.g., `require_termination = data[0] == '1' ? 1 : 0`), but testing invalid values like -1 or large integers could reveal error handling flaws or undefined behaviors.

Our Solution. To address these limitations, we propose to enhance existing fuzz drivers by introducing independent option parameter control mechanisms. Our approach works by augmenting the original driver with option-aware code that implements an adaptive option parameter mutation strategy, as shown in the code example:

- *Coverage-Guided Option Selection*: Rather than relying on random option selection, MUTATO adopts a coverage-guided strategy in which option mutations are initiated only when coverage growth stagnates (e.g., after 10,000 executions or at 5-second intervals, as implemented in the `check_coverage_growth` function). When edge coverage continues to increase, the system retains the current option value with high probability (95%, as determined by the epsilon parameter), thereby enabling in-depth exploration of promising execution paths.
- *Comprehensive Option Space Exploration*: MUTATO examines both valid and invalid option values during fuzzing. As illustrated by the `get_new_require_termination` function, the approach evaluates a range of values, including legal options `(0, 1)` as well as boundary and illegal values `(-1, 2)`. This comprehensive exploration ensures robust assessment of the API's behavior under both expected and unexpected configurations.

Benefits. Our design offers several key advantages:

- *Universal Fuzzer Compatibility*: By implementing option control within the fuzz driver itself (as shown in the `LLVMFuzzerTestOneInput` function), MUTATO achieves broad compatibility with different fuzzing engines (e.g., AFL++, LibFuzzer) without requiring modifications to the fuzzer core.
- *Input Format Preservation*: Unlike approaches that embed options in input data, our implementation maintains separate tracking of options (via `new_require_termination`), allowing the original input format to remain intact while still controlling API options.
- *Comprehensive Option Testing*: The system methodically explores diverse option values through the array of options in `get_new_require_termination`, ensuring thorough testing of the API's behavior under various configurations.
- *Efficient Option Space Exploration*: Through the coverage-based feedback mechanism in `check_coverage_growth`, MUTATO balances exploration and exploitation, ensuring that promising option values receive extended exploration time while still periodically testing new options to discover additional vulnerabilities. This function can be customized based on the specific fuzzer type (e.g., AFL++ or LibFuzzer), allowing MUTATO to leverage the native coverage tracking mechanisms of each fuzzer while maintaining a consistent option mutation strategy.

IV. MUTATO APPROACH

In this section, we present our approach to adaptive fuzzing with option parameters. We first introduce a domain-specific specification language that enables the systematic enhance-

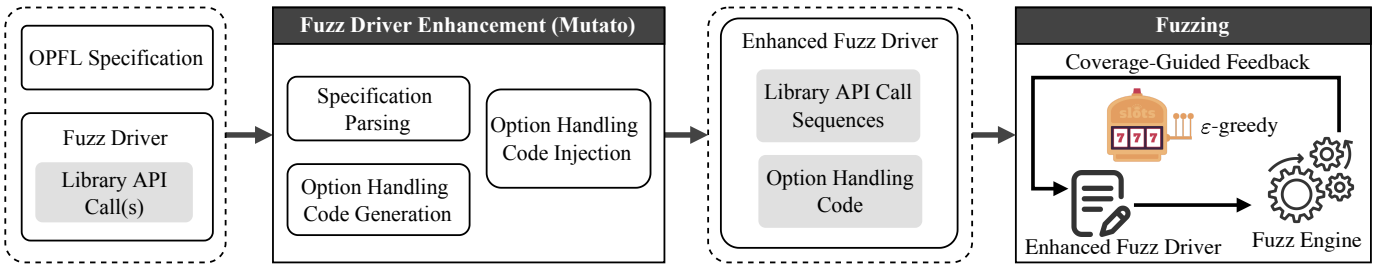


Fig. 1: Overview of MUTATO.

```
Spec ::= "Target" IDN
         OptionGroupBlock+
         ConstraintBlock
         SelectBlock FeedbackBlock MutateBlock

OptionGroupBlock ::= "OPGroup" IDN "{" OptionField + "}"
OptionField      ::= IDN":"Type ["=" DefaultValue]

ConstraintBlock  ::= "Constraint" "{" ConstraintExpr + "}"
ConstraintExpr   ::= FOLExpr
/* First-order logic expression over OptionFields */

FeedbackBlock    ::= "Feedback" "{" FeedbackExpr "}"
FeedbackExpr     ::= FeedbackName "(" ParamList? ")"
FeedbackName     ::= "coverage" | ...

SelectBlock      ::= "Select" "{" SelectExpr "}"
SelectExpr       ::= SelectName "(" ParamList? ")"
SelectName       ::= "epsilon_greedy"
                   | "random_search" | ...

ParamList        ::= Param("," Param)*
Param            ::= IDN "=" Literal

MutateBlock      ::= "Mutate" "{" MutateExpr + "}"
MutateExpr       ::= MutateName "(" IDN ")"
MutateName       ::= "select_from_set" | "enum_switch"
                   | "bitflip" | "arith" | ...

IDN              ::= letter { letter | digit }
Type             ::= "bool" | "choice" | "numeric"
                   | "string"
DefaultValue     ::= Literal

Literal          ::= BoolLiteral /* true or false */
                   | ChoiceLiteral /* user-defined choice */
                   | NumericLiteral /* A concrete numeric value */
                   | StringLiteral /* A concrete string value */
```

Fig. 2: EBNF-like grammar for our option parameter fuzzing language (OPFL).

ment of existing fuzz drivers, facilitating comprehensive exploration of option parameter spaces (§IV-A). Next, we elaborate on our methodology for option parameter fuzzing, which leverages a coverage-guided feedback mechanism and an adaptive option selection strategy to maximize test coverage and uncover vulnerabilities in target libraries (§IV-C). Finally, we describe the code instrumentation process that transforms existing fuzz drivers into option-aware fuzzing drivers (§IV-D) based on our specification language. Figure 1 provides an overview of the MUTATO framework, which accepts an existing fuzz driver as input and automatically generates an enhanced, option-aware fuzz driver.

A. Option Parameter Fuzzing Language

We introduce a domain-specific specification language, option parameter fuzzing language (OPFL), designed to formally express strategies for producing enhanced, option-aware fuzz drivers from existing ones. OPFL provides a structured and expressive mechanism for defining how fuzzing options should be grouped, constrained, selected, and mutated, thereby enabling systematic exploration of the configuration space. The language is both declarative and semantically rich, allowing users to precisely guide the enhancement process while abstracting away low-level implementation details. The formal grammar of the language is presented in Figure 2.

A complete specification, denoted by the non-terminal *Spec*, begins with the declaration of the target fuzz driver via the `"Target"` keyword, followed by a series of *OptionGroupBlocks* which define the configuration parameters relevant to the fuzz driver. Optional blocks for constraints (*ConstraintBlock*), selection strategies (*SelectBlock*), feedback mechanisms (*FeedbackBlock*), and mutation strategies (*MutateBlock*) can be included to further refine the enhancement process.

Each *OptionGroupBlock* begins with `"OPGroup"` keyword followed by a unique identifier (*IDN*) and a set of *OptionField*s enclosed in braces. An *OptionField* defines a configuration parameter by specifying its name (*IDN*), type (*Type*), and optionally a default value. Constraints over these option fields are expressed using the *ConstraintBlock*, which contains one or more *ConstraintExpr* entries. Each *ConstraintExpr* is a first-order logic expression (*FOLExpr*) defined over the declared *OptionField*s. These constraints serve to define the space of option values to be explored during fuzzing, allowing for both semantically valid combinations and potentially invalid but type-compatible ones. The constraints can capture dependencies and mutual exclusivity relationships among option parameters within the same API or across different APIs, with the constraint strictness determining the balance between exploring valid parameter combinations versus investigating potential boundary conditions and error states.

The *FeedbackBlock* defines how runtime information is used to inform the option selection process. A *FeedbackExpr* specifies a feedback mechanism (*FeedbackName*). Feedback parameters can be fine-tuned via the optional *ParamList*, enabling adaptive fuzzing strategies based on execution feedback. The *SelectBlock* allows users to specify a selection strategy for navigating the configuration space. Each *SelectExpr* defines a search algorithm (*SelectName*) and optionally a

Algorithm 1: Adaptive Option Selection During Fuzzing

Input: Option space $O_i = \{o_1, o_2, ..., o_n\}$, Epoch size N, Exploration rate ϵ
Output: Optimized sequence of option values

$o_{curr} \leftarrow$ Random option from O_i
$C_{prev} \leftarrow 0$
$C_{curr} \leftarrow 0$
while *fuzzing budget not exhausted* **do**
  `/* Fixed-Option Phase */`
  for $i \leftarrow 1$ **to** B **do**
    `/* B can be N executions or time limit T */`
    Generate mutation with current option o_{curr}
    Execute target program
    $C_{curr} \leftarrow$ `ComputeCurrentCoverage()`
  `/* Decision Phase */`
  if $C_{curr} > C_{prev}$ **then**
    `/* Coverage improved with current option */`
    $r \leftarrow$ Random value in $[0, 1]$
    if $r < \epsilon$ **then**
      `/* Explore: try different option */`
      $o_{curr} \leftarrow$ Random option from $O_i \setminus \{o_{curr}\}$
    `/* Else: exploit by keeping current option */`
  else
    `/* Coverage stagnated */`
    $o_{curr} \leftarrow$ Random option from $O_i \setminus \{o_{curr}\}$
  $C_{prev} \leftarrow C_{curr}$

list of parameters (*ParamList*). The detailed option parameter mutation of MUTATO is described in Section IV-C. Finally, the *MutateBlock* specifies how option values should be mutated. Each *MutateExpr* defines a mutation strategy (*MutationName*) and its parameters. This allows for flexible and customizable mutation strategies that can be tailored to the specific characteristics of the target API.

B. Integration of OPFL Specification

To apply the OPFL specification to fuzzing, MUTATO analyzes the existing fuzz driver to identify API calls matching those defined in the `"Target"` and *OptionGroupBlock*s of the OPFL specification. For each matched API call, MUTATO generates option parameter values based on the *ConstraintBlock*, which defines valid values and invalid values, as well as constraints between the option and other options (if any), including dependencies and mutual exclusivity among different parameters. Based on the selection strategies (*SelectBlock*), feedback mechanisms (*FeedbackBlock*), and mutation strategies (*MutateBlock*) defined in the OPFL specification, MUTATO generates specific logic and embeds it into the original fuzz driver. Then this enhanced, option-aware driver can simultaneously explore both input and option spaces during the fuzzing process.

C. Adaptive Option Parameter Mutation

Based on the feedback mechanism defined in the *FeedbackBlock*, the enhanced driver can dynamically adjust option values during fuzzing according to the defined feedback metric. In the following discussion, we focus on coverage-guided fuzzing as the default fuzzing method, where increases in code coverage (e.g., new branches or paths) serve as the principal metric for assessing the effectiveness of generated inputs in exercising new program behaviors. Building upon this foundation, we extend the application of coverage metrics to inform the selection of option parameters. Rather than selecting option values randomly for each execution, our approach employs a structured algorithm that adaptively chooses option parameters based on their observed contribution to coverage improvement. This strategy systematically balances the exploitation of option values that have demonstrated effectiveness with the exploration of alternative values, thereby maximizing the likelihood of uncovering previously untested behaviors.

Let $\text{Cov}_i^t : O_i \rightarrow \mathbb{R}^+$ denote the coverage state at time t, representing code coverage metrics achieved when using option parameter $o \in O_i$. We maintain a global coverage snapshot C_{prev} to track coverage progress for each active option. As shown in Algorithm 1, the fuzzing process operates in discrete epochs with two distinct phases (ℓ_5-ℓ_8):

Fixed-Option Phase. During this phase, we keep the option parameter fixed while only mutating the input data. This approach is more efficient than changing option values after each execution, as it:

- Allows the fuzzer to thoroughly explore input space with the current option value o_{curr}.
- Performs consecutive input mutations within a predefined boundary condition $B : m_n = \mu^n(m_{n-1})$ where B can be either temporal (e.g., $t \leq T$ seconds) or quantitative (e.g., $n \leq N$ executions).
- Maintains a mutation counter n and the initial coverage snapshot C_{prev}.
- Executes $a_i(m_n, o_{\text{curr}})$ without intermediate coverage comparisons.

where m_n denotes the input data at the n-th mutation, and μ^n represents the mutation function applied at this step. This approach enables the fuzzer to systematically explore the input space under a fixed option value.

Decision Phase. Upon reaching the execution boundary condition B with a fixed option value, we evaluate the coverage improvement ΔC and determine the subsequent option selection strategy (ℓ_9-ℓ_{15}). Specifically:

- We compare the current coverage $C_{\text{curr}} = \text{Cov}_i^t(o_{\text{curr}})$ with the previous coverage snapshot C_{prev}.
- The coverage gain is computed as $\Delta C = |C_{\text{curr}} \setminus C_{\text{prev}}|$.
- The next option value is selected according to the following rule:

$$o_{\text{next}} = \begin{cases} \text{Random}(O_i \setminus \{o_{\text{curr}}\}) & \text{if } \Delta C = 0 \\ \epsilon\text{-Greedy}(O_i, o_{\text{curr}}) & \text{if } \Delta C > 0 \end{cases}$$

This means, if no new coverage is observed ($\Delta C = 0$), we select one or more option values from the option space O_i using the strategy specified in the *SelectBlock* and ensure that the selected option(s) satisfy constraints defined

in the *ConstraintBlock*. Conversely, if coverage increases ($\Delta C > 0$), we apply the ϵ-greedy strategy specified in the *MutateBlock*, balancing continued exploitation of the current option with exploration of alternative options.

This two-phase approach balances thoroughness with efficiency, avoiding the overhead of checking coverage after every execution while still adapting to coverage feedback. By batching executions with the same option value, we reduce the computational overhead of coverage comparison and option selection, which can be significant in high-throughput fuzzing.

We adopt the ϵ-greedy algorithm because it efficiently balances exploitation of productive option values with exploration of alternatives, preventing the fuzzer from getting stuck in local optima. When coverage increases, the algorithm keeps the current productive option with probability $1 - \epsilon$ (exploitation) and randomly selects a different option with probability ϵ (exploration).

For example, with $\epsilon = 0.1$, when an option value successfully increases coverage, we continue using it 90% of the time while occasionally (10% of the time) trying alternative options. This ensures the fuzzer spends most time on productive configurations while maintaining enough exploration to discover new code paths that other option values might unlock.

To illustrate our complete approach with a concrete example, consider fuzzing the cJSON library with the `require_termination` option shown in Listing 1. While the API documentation defines this as a boolean parameter $O_i = \{0, 1\}$ which is actually an `int` type. This creates an interesting constraint space where we can explore both semantically valid values (0, 1) and type-compatible but potentially unintended values (-1, 2) to test error handling paths:

1) We start by setting the default option value $o_{\text{curr}} = 1$ (require termination).
2) During the fixed-option phase, we execute the fuzzer with this option value.
3) Every 5 seconds (boundary condition), we check if coverage has increased.
4) If coverage increases, using ϵ-greedy with $\epsilon = 0.05$, we have a 95% chance of keeping option 1 (since it's productive) and a 5% chance of trying option 0.
5) If coverage stagnates after a check period, we would randomly switch to another option value.
6) Although the semantic constraint is $O_i = \{0, 1\}$, we also test type-compatible but invalid values (-1, 2, 255) to explore how the library handles these unintended inputs.
7) When we try a different option value, we might discover that it reaches different code paths, particularly error handling code, increasing coverage.
8) This process continues adaptively, spending more time on productive option values while still maintaining the ability to explore alternatives.

This approach efficiently navigates the option parameter space by spending more time on productive option values while still maintaining the ability to explore alternatives when progress stalls. The key insight is that not all option values are equally valuable for discovering bugs or increasing coverage, and our strategy adaptively learns which options are more promising during the fuzzing process itself.

D. Option-Aware Fuzz Driver Enhancement

OPFL (§IV-A) provides a formal foundation for the automated enhancement of option-aware fuzz drivers. Given a user-provided OPFL specification, the enhancement process proceeds as follows. First, the specification is parsed to extract the target fuzz driver, option groups, constraints, and the associated feedback, selection, and mutation strategies. For each option group, the enhancement engine generates code to declare and initialize option parameters, incorporating type checks and default values as specified. Constraints are systematically enforced through the insertion of runtime checks or static assertions, ensuring that only semantically valid option combinations are exercised during fuzzing. The feedback and selection strategies are mapped to corresponding code modules, enabling adaptive option selection based on runtime metrics such as code coverage. Mutation strategies are realized as dedicated mutator functions, facilitating systematic exploration of the option parameter space. Finally, the original fuzz driver is instrumented with the enhanced logic, resulting in an enhanced, option-aware fuzzing engine capable of jointly exploring both input data and configuration options in a principled and automated manner.

Example 1: Consider the following specification for the cJSON library fuzz driver:

```
Target: cjson_read_fuzzer
OPGroup ParseOptions {
    require_termination: bool = true
}
Constraint {
    // Allow both valid boolean values
    // and invalid type-compatible values
    (require_termination == 0) ||
    (require_termination == 1) ||
    (require_termination == -1) ||
    (require_termination == 2)
}
Feedback {
    coverage(branches=true)
}
Select {
    epsilon_greedy(epsilon=0.05)
}
Mutate {
    select_from_set([0, 1, -1, 2])
}
```

Given this specification, the enhancement engine will:

- Generate code to declare the `require_termination` parameter with appropriate type checks.
- Allow both semantically valid values (0, 1) and type-compatible but potentially unintended values (-1, 2) to test error handling.
- Integrate an ϵ-greedy selection strategy with $\epsilon = 0.05$, guided by branch coverage feedback.
- Implement mutation operators that select from predefined values for `require_termination`.

TABLE I: Basic information of the 10 libraries. API(T) denotes the total number of APIs. API(O) means the number of APIs with option parameters. API(U) is the number of APIs exercised by fuzz drivers.

Library	Version	LoC	Branch	API(T)	API(O)	API(U)	Time(min)
cJSON	1.7.18	10K	1.4K	76	6	6	15
lcms	2.17	46K	8.1K	218	16	7	25
libpcap	1.10.5	50K	7.8K	67	14	5	18
lunasvg	3.2.1	25K	8.3K	33	6	5	12
libtiff	4.7.0	91K	16.3K	172	4	4	10
utf8proc	2.10.0	19K	1.5K	36	8	5	14
liblouis	3.33.0	37K	6.6K	34	9	5	16
xlsxio	0.2.35	11K	1.9K	45	4	4	8
libzip	1.11.3	18K	4.2K	113	29	8	22
libvpx	1.14.0	367K	17.6K	7	31	6	20
Total	-	674K	73.7K	825	103	55	160

- Inject the generated logic into the original `cjson_read_fuzzer`, producing an enhanced driver similar to Listing 1.

This automated enhancement process allows users to rapidly generate robust, option-aware fuzz drivers tailored to the semantics of the target API, without manual code modifications. OPFL thus bridges the gap between high-level fuzzing strategies and low-level driver implementation, enabling scalable and systematic exploration of complex configuration spaces.

V. Evaluation

In this section, we evaluate MUTATO to answer the following research questions:

- **RQ1**: How effective is MUTATO at enhancing existing fuzz drivers in terms of code coverage compared to the original drivers generated by baseline tools?
- **RQ2**: Can MUTATO-enhanced drivers discover bugs that are not found by the original drivers, particularly those triggered by specific option parameter values?
- **RQ3**: How does the performance of MUTATO compare with that of prefix-based option-encoding methods?
- **RQ4**: What are the impacts of the design choices of MUTATO on its effectiveness?

A. Experimental Setup

1) Target Libraries: We evaluated MUTATO on 10 widely-used C/C++ libraries across diverse application domains: `cJSON` [22], `lcms` [23], `liblouis` [24], `libpcap` [25], `libtiff` [26], `libvpx` [27], `libzip` [28], `lunasvg` [29], `utf8proc` [30], and `xlsxio` [31]. These libraries represent a diverse set of widely-adopted open-source components across various domains including network protocols, JSON parsing, image processing, compression, text processing, and file format handling. Most of these libraries are included in OSS-Fuzz [32] project, which continuously fuzzes critical open-source software. Notably, these libraries contain many APIs with option parameters, making them suitable candidates for our evaluation. The basic statistics of the tested libraries are summarized in Table II, including their versions, lines of code, number of branches, total number of APIs (API-T), the number of APIs with option parameters (API-O), and the subset of those APIs that were actually used in our evaluation (API-U).

2) Baseline: Since MUTATO enhances original fuzz drivers by mutating option parameters, we compare it against two categories of baselines:

- **Original Fuzz Drivers:** We use original drivers generated by two SOTA driver generation tools, CKGFuzzer [4] (**CKG**) and OSS-Fuzz-Gen [5] (**OFG**)—without applying any manual changes or fixes, and exclude drivers that fail to compile or execute properly, as our primary baselines.
- **Prefix-based Option Encoding Method:** ConfigFuzz [10] uses a prefix-based option encoding method to encode program option parameters as additional bytes at the beginning of the input, allowing dynamic variation of options during fuzzing by processing these prefix bytes within the driver. Since ConfigFuzz is not open-sourced, we re-implemented this baseline following their methodology. Our implementation incorporates the key strategies as follows:
  - *Prefix-based Encoding:* We prepend a small number of bytes to the fuzzer's input, following the exact encoding scheme described in [10]. These prefix bytes are used to select different options through direct indexing.
  - *Representative Value Selection:* While ConfigFuzz requires manual extraction of representative values from documentation, we employ the same LLM-manual hybrid strategy used in our MUTATO tool for efficiency (see Section V.A.3 for details). For options with finite choices (e.g., enums), we enumerate all possible values. For options with infinite ranges (e.g., integers), we select representative values including boundary conditions (e.g., 0, 1, INT_MAX), commonly used values, and known problematic values.
  - *Driver Implementation:* We implement a pre-processing stage in the driver that reads and decodes the prefix before the main API call, similar to the approach in ConfigFuzz.

  We adapt this technique for API drivers by modifying the CKG/OFG-generated drivers to incorporate these mechanisms. These enhanced drivers are denoted as **CKG(∗)** and **OFG(∗)**, respectively.

The CKG and OFG drivers modified by MUTATO are denoted as **CKG(+)** and **OFG(+)**, respectively. It is important to emphasize that, whether CKG(∗)/OFG(∗) drivers or CKG(+)/OFG(+) drivers, all only modify the option parameter handling within the CKG and OFG drivers. These changes do not affect any other parts of the drivers, ensuring their core functionality remains unchanged.

3) Writing OPFL Specifications: To obtain the API OPFL specification for each library, we first used large language models (LLMs) with OPFL templates, example specifications, and header files as inputs to automatically generate initial specifications covering both valid and invalid options. These generated specifications then undergo manual verification and adjustment to ensure accuracy. The manual effort primarily involves identifying user-controllable option parameters from library APIs and determining their valid value ranges from public header files and API documentation. Invalid values are derived using standard boundary testing techniques. For

TABLE II: Coverage of original CKGFuzzer and OSS-Fuzz-Gen (OFG/CKG) generated drivers, prefix-based option encoding method (OFG(∗)/CKG(∗)), and MUTATO's option-enhanced OFG(+)/CKG(+) using AFL++ and LibFuzzer across 10 libraries.

Library	AFL++						Libfuzzer						Exec/Sec		
	OFG	OFG(∗))	OFG(+)	CKG	CKG(∗))	CKG(+)	OFG	OFG(∗))	OFG(+)	CKG	CKG(∗))	CKG(+)	OFG/ CKG	OFG(∗))/ CKG(∗))	OFG(+)/ CKG(+)
cJSON	221	255(15%)	257(16%)	276	345(25%)	347(26%)	186	243(31%)	249(34%)	325	334(3%)	334(3%)	5963	8395	7656
lcms	793	901(14%)	910(15%)	640	768(20%)	766(20%)	547	632(16%)	642(17%)	577	643(11%)	652(13%)	4908	4878	4996
libpcap	1281	934(-27%)	1651(29%)	1670	1513(-9%)	1786(7%)	1453	1125(-23%)	1597(10%)	1271	1197(-6%)	1340(5%)	6593	6539	6588
lunasvg	1736	1254(-28%)	1782(3%)	1089	1247(15%)	1589(46%)	1134	1074(-5%)	1197(6%)	1040	1008(-3%)	1159(11%)	3199	3088	2976
libtiff	857	769(-10%)	951(11%)	1078	870(-19%)	1159(7%)	1213	1170(-4%)	1457(20%)	929	834(-10%)	1143(23%)	7156	7508	7548
utf8proc	28	78(179%)	125(345%)	39	128(231%)	130(236%)	39	61(56%)	59(51%)	36	121(236%)	124 (244%)	26984	22295	23415
liblouis	554	571(3%)	579(4%)	221	242(10%)	252(14%)	561	593(6%)	604(8%)	321	340(6%)	342(7%)	5093	4751	4778
xlsxio	1079	1213(12%)	1250(16%)	1402	1529(19%)	1535(9%)	82	94(15%)	98(20%)	172	189(10%)	191(11%)	4423	5002	3496
libzip	503	451(-10%)	583(16%)	720	564(-22%)	955(33%)	489	476(-3%)	611(25%)	834	775(-7%)	848(2%)	12126	11969	11643
libvpx	2437	2443(0.2%)	2451(1%)	1538	1542(0.2%)	1589(3%)	1438	1443(0.3%)	1451(1%)	1073	1274(19%)	1361(27%)	1080	976	1002
Total	9488	8869(-7%)	10538(11%)	8672	8748(1%)	10106(17%)	7142	6911(-3%)	7965(12%)	6578	6585(2%)	7494(14%)	7752	7540	7410

other configuration parameters, we use empirically determined default values, such as setting the exploration rate to $\epsilon = 0.05$. The time required to obtain OPFL specifications for each library (in minutes) is detailed in Table I (*Time* Column). These reported times (8-25 minutes) represent the effort of developers with general programming experience, as the task primarily involves straightforward API documentation review and does not require specialized expertise with MUTATO.

For the valid and invalid values of option parameters defined in the specifications, we employ the following strategies based on option type (These valid and invalid values are also reused in the Prefix-based Option Encoding Method): (1) For enumerable option types, such as Boolean options and Enumerated choices, we leverage their finite nature by identifying all possible valid and invalid values. (2) For non-enumerable option types, specifically Numeric options and String options, where the value space can be extremely large or infinite, a different strategy is employed. We apply principles from software testing's boundary testing to select a representative set of both valid and invalid values.

4) Implementation: We implemented MUTATO using Tree-Sitter [33], a parser generator framework that enables precise analysis and modification of baseline fuzz drivers. Following the driver synthesis methodology described in Section IV-D, our implementation systematically identifies APIs with option parameters in the baseline drivers, replaces the original hard-coded option values with code for dynamic option value generation, and inserts instrumentation for coverage monitoring to guide the option selection process.

We evaluated MUTATO with both AFL++ and LibFuzzer to demonstrate its fuzzer-agnostic design and to assess performance improvements across different fuzzing engines. Since both CKG and OFG generate LibFuzzer-compatible drivers by default, we adapted these drivers for AFL++ by implementing wrapper functions that provide a standard `main` entry point, which internally invokes `LLVMFuzzerTestOneInput` with the fuzzer-supplied inputs. In addtion, although MUTATO produces fuzzer-independent drivers, certain adaptations were necessary to accommodate the specific requirements of different fuzzing engines. Therefore, we implemented distinct coverage data collection mechanisms for AFL++ and LibFuzzer to ensure effective guidance of our option selection strategy. In addition, as described in the *Fixed-Option* Phase of Section IV-C, the enhanced drivers do not query coverage every time but instead use an interval-based strategy to improve efficiency. After each interval, the newly collected coverage is compared with the previous coverage to determine whether an option change has led to increased code coverage. Specifically, for AFL++ enhanced drivers, we query coverage every 5 seconds, while for LibFuzzer enhanced drivers, we collect coverage every 10,000 executions to decide whether to change the options. Crucially, all such modifications were confined to the driver level, without necessitating any changes to the underlying fuzzing engines themselves. This design ensures full compatibility with standard fuzzing tools and workflows, allowing seamless integration of MUTATO into existing fuzzing pipelines without disrupting established testing processes. Section IV-C

5) Evaluation Environments: All experiments were conducted on a server equipped with two AMD EPYC 9254 24-Core Processors (48 cores total, 96 threads with hyperthreading) and 1TB of RAM. The system runs on a 64-bit version of Debian GNU/Linux 12. GPT-4o was used to generate fuzz drivers for both CKG and OFG, as both of them leveraging large language models in their driver generation process.

Each fuzzing campaign ran for 24 hours to ensure sufficient time for exploration of the option space. To handle LibFuzzer's default behavior of terminating upon crash discovery, the fuzzing process is set to automatically restart to ensure continuous execution for 24 hours, even after crashes are found. To account for the randomness inherent in fuzzing, we repeated each experiment ten times and report the average results. For all experiments, we used the same seed corpus consisting of minimal valid inputs for each library to ensure fair comparison between baseline and option-enhanced approaches.

B. Code Coverage (RQ1)

Table II presents the coverage results of our evaluation across the 10 libraries using different fuzz drivers (CKG and OFG) and fuzzing engines (AFL++ and LibFuzzer). We compare the performance of the original drivers generated

by OFG and CKG against their option-enhanced versions created by MUTATO (OFG(+) and CKG(+)). For AFL++, we measure coverage in terms of edges, while for LibFuzzer, we use branches as the coverage metric. This difference in metrics stems from the underlying instrumentation mechanisms: AFL++ uses edge coverage (transitions between basic blocks) to track execution paths, while LibFuzzer's SanitizerCoverage instrumentation primarily reports branch coverage (individual control-flow decisions).

The results in Table II demonstrate overall coverage improvements when enhancing fuzz drivers with option parameter exploration. For AFL++, the option-enhanced drivers consistently outperform their baseline counterparts across all libraries. With OFG drivers, MUTATO (OFG(+)) achieves an average coverage increase of 11%, with the most substantial improvements in `utf8proc` (3.45×), `libpcap` (0.29×), and `xlsxio` (0.16×). When applied to CKG drivers, MUTATO (CKG(+)) achieves a higher average coverage increase of 17%, with `utf8proc` (2.36×), `lunasvg` (0.46×), and `libzip` (0.33×) showing the largest coverage gains.

For LibFuzzer, similar patterns emerge with consistent coverage gains across all tested libraries. The OFG (+) drivers achieve an average coverage increase of 13%, with `utf8proc` (0.51×), `cJSON` (0.34×), and `libzip` (0.25×) showing the largest improvements for OFG fuzz drivers. With CKG drivers, CKG (+) delivers an average coverage increase of 14%, with `utf8proc` (2.44×), `libvpx` (0.27×), and `libtiff` (0.23×) demonstrating the most significant gains.

These consistent improvements stem from MUTATO's approach of building upon the original drivers generated by OFG and CKG. By preserving all functionality of the original drivers while adding dynamic option exploration capabilities, MUTATO ensures that all previously covered code paths remain accessible while significantly increasing the probability of discovering new paths through option variation. In addition, the execution speed remains largely unaffected by MUTATO's enhancements. As shown in the Exec/Sec measurements of Table II, MUTATO maintains comparable performance with only a modest 4.4% decrease (7,410 vs 7,752 executions per second on average) demonstrating that the dynamic option exploration introduces negligible computational overhead while delivering substantial coverage improvements. Although the fuzzing process inherently involves randomness, our results demonstrate that systematically exploring the option space consistently leads to better coverage across different libraries and fuzzing engines. The box plots in Figures 5 and 6 further illustrate the coverage distribution across multiple runs for AFL++ and LibFuzzer respectively, showing that the option-enhanced drivers consistently achieve higher coverage with less variance compared to the baseline approaches.

The coverage trends in Figures 3 and 4 (Appendix) demonstrate that OFG(+)/CKG(+) consistently outperform their baseline counterparts (OFG/CKG) in coverage growth over the 24-hour fuzzing period. Table VI in Appendix presents the statistical significance of these improvements using the Mann-Whitney U test and A12 effect size [34]. For AFL++, MUTATO ($p < 0.05$) improves coverage in 90% (9/10) of libraries compared to the original OFG drivers, and in 100% (10/10) of libraries compared to CKG drivers. For LibFuzzer, MUTATO achieves statistically significant improvements in 90% (9/10) of libraries for both OFG and CKG drivers. The A12 effect size values, mostly above 0.7, further indicate that the coverage improvements are not only statistically significant but also substantial in magnitude.

The coverage improvement may vary between libraries, reflecting differences in their API design and implementation. Libraries showing dramatic coverage increases typically have option parameters that significantly alter execution paths through function dispatching patterns. For example, the library API `utf8proc_reencode(utf8proc_int32_t *buffer, utf8proc_ssize_t length, utf8proc_option_t options)` in the library `utf8proc` reveals that different `options` values trigger entirely different code paths by dispatching to distinct encoding routines like `charbound_encode_char` or `utf8proc_encode_char`, with minimal shared code between these paths. This pattern of option-driven function dispatching explains the library's exceptional coverage gains (3.45× with AFL++). In contrast, libraries like `libvpx` (0.01% with OFG drivers) show small coverage gains because their option parameters, such as the `align` parameter in `vpx_img_alloc()`, mainly affect performance rather than execution flow. These options typically result in simple conditional checks without introducing new code paths.

Answer to RQ1: Systematic exploration of option parameters significantly increases code coverage compared to traditional fuzzing with fixed option values. This improvement is consistent across libraries, driver tools, and fuzzing engines, demonstrating the effectiveness and generality of MUTATO's adaptive option mutation strategy.

C. Bug Detection (RQ2)

We further evaluated MUTATO's effectiveness in discovering real-world vulnerabilities. Table III quantifies our findings by presenting the number of unique vulnerabilities (UV) discovered by each approach, highlighting exclusive vulnerabilities (EV) that were only detected by a specific technique, and providing detailed information about the vulnerability types and current remediation status across the evaluated libraries.

The evaluation revealed that MUTATO (OFG(+)/CKG(+)) outperforms original fuzzing drivers (OFG/CKG) in vulnerability discovery. As shown in Table III, MUTATO discovered a total of 12 unique vulnerabilities across both AFL++ and LibFuzzer, compared to only 5 vulnerabilities found by the baseline approaches. Notably, 7 vulnerabilities were exclusively found by MUTATO. The discovered vulnerabilities span critical security issues, including heap buffer overflows (HBO), stack buffer overflows (SBO), null pointer dereferences (NPD), memory leaks (ML), and use of uninitialized variables (UUV).

It also shows that MUTATO encompasses all vulnerabilities discovered by baseline approaches, whether they were found by OFG or CKG. This is expected since MUTATO is an

TABLE III: The vulnerabilities found by original CKG-Fuzzer and OSS-Fuzz-Gen (OFG/CKG) generated drivers, prefix-based option encoding method (OFG(∗)/CKG(∗)), and MUTATO's option-enhanced OFG(+)/CKG(+) drivers using AFL++ and LibFuzzer. (UV: Unique vulnerability, EV: Exclusive vulnerability; HBO: Heap Buffer Overflow, UUV: Use of Uninitialized Variable, NPD: Null Pointer Dereference, ML: Memory Leak, SBO: Stack Buffer Overflow)

Library	OFG/ CKG	OFG(∗)/ CKG(∗)	OFG(+)/ CKG(+)	Vulnerability Type	Status	Option Type
	UV/EV	UV/EV	UV/EV			
lcms	0/0	1/0	1/0	HBO	Fixed	Invalid
lunasvg	1/0	1/0	1/0	HBO	Fixed	Valid
	0/0	1/0	1/0	UUV (CVE-xxx56)	Confirmed	Invalid
libtiff	1/0	1/0	1/0	NPD	Confirmed	Invalid
	0/0	0/0	1/1	ML	Reported	Valid
utf8proc	1/0	1/0	1/0	HBO	Confirmed	Valid
	0/0	1/0	1/0	HBO	Confirmed	Valid
	0/0	0/0	1/1	HBO	Confirmed	Valid
liblouis	0/0	1/0	1/0	SBO	Fixed	Invalid
xlsxio	1/0	1/0	1/0	NPD (CVE-xxx49)	Confirmed	Valid
	1/0	1/0	1/0	ML	Reported	Invalid
libzip	0/0	0/0	1/1	NPD (CVE-xxx55)	Fixed	Valid
Count	5/0	9/0	12/3	-	-	-

enhancement built on top of the original fuzzing drivers, preserving their core functionality while extending their capabilities. By retaining the original source code and adding option-aware fuzzing mechanisms, MUTATO maintains the vulnerability detection capabilities of the baseline approaches while discovering significantly more vulnerabilities that would otherwise remain hidden when using fixed option values.

Of the 12 vulnerabilities identified, 7 were triggered by valid option parameters, including two with CVEs: CVE-xxx49 in xlsxio and CVE-xxx55 in libzip. These vulnerabilities arise when legitimate option values trigger bugs in under-tested code paths. Specifically, CVE-xxx49 in xlsxio arises during XML parsing, causing a null pointer dereference crash due to untested valid option paths. CVE-xxx55 in libzip also involves a null pointer dereference triggered only when specific valid option parameters are enabled, exposing a flaw in the handling of non-default configurations. The remaining 5 vulnerabilities were triggered by invalid option parameters, including one with a CVEs: CVE-xxx56 in lunasvg. These invalid option vulnerabilities arise from insufficient validation of option parameters, causing crashes when unexpected values are processed. Specifically, CVE-xxx56 in lunasvg occurs during SVG rasterization, leading to a segmentation fault crash due to an out-of-bounds memory access triggered by an invalid option parameter that lacks proper validation. These vulnerabilities were missed by fuzzing methods that do not explore option configurations systematically, leaving uncommon valid options and invalid option values unexplored.

In addition, the distribution of vulnerabilities across libraries correlates with our coverage improvement results. Libraries that showed the most significant coverage improvements with MUTATO, such as `utf8proc` and `lunasvg`, also yielded vulnerabilities that were not found by baseline approaches. This confirms that improved code coverage through systematic option parameter exploration translates directly to enhanced vulnerability discovery.

Answer to RQ2: MUTATO's option-aware fuzzing uncovers more real-world vulnerabilities than traditional fuzzing, finding all baseline bugs plus seven additional ones. Systematic exploration of option parameters is essential for exposing security issues that fixed option values miss, as both valid and invalid options can trigger unique vulnerabilities.

D. Prefix-based Option Encoding Method Comparison (RQ3)

To further evaluate the effectiveness of MUTATO, we compared MUTATO (CKG(+)/OFG(+)) against a prefix-based option encoding method CKG(∗) and OFG(∗). This baseline approach encodes option parameters as additional bytes at the beginning of the input, allowing the fuzzer to explore different option values by mutating these prefix bytes.

For code coverage, MUTATO consistently outperforms the prefix-based method across all 10 libraries with both AFL++ and LibFuzzer. Table II shows that MUTATO's OFG(+) and CKG(+) drivers achieve average coverage increases of 11% and 17% for AFL++ and 12% and 14% for LibFuzzer over the original OFG and CKG drivers, respectively. In contrast, the prefix-based method struggles, with OFG(∗) showing a 7% coverage decrease and CKG(∗) a marginal 1% increase for AFL++, and a 3% decrease for OFG(∗) and a 2% increase for CKG(∗) for LibFuzzer compared to the originals. Notably, the prefix-based method performs even poorly in libraries with complex input formats than original OFG and CKG generated drivers, such as `libpcap` and `lunasvg`, and only performs comparably with CKG(∗) and OFG(∗) in simpler cases like `cJSON`. This is because the prefix-based method merges options and data into a single input, causing fuzzers like AFL++ to treat them as equivalent domains. For instance, AFL++'s *auto_extra* feature may mistakenly treat prefix bytes as dictionary tokens for file content mutations, leading to ineffective mutations and reduced coverage in complex input formats. MUTATO avoids this by treating option values and main content as distinct mutation domains, using coverage-guided adaptive mutations to systematically explore option-driven paths, resulting in consistent coverage gains with only a 1.6% reduction in execution speed (7410 vs. 7540 executions/second for OFG(+)/CKG(+) and OFG(∗)/CKG(∗)).

In vulnerability detection, MUTATO also performs better than the prefix-based method, as shown in Table III. MUTATO's OFG(+)/CKG(+) drivers uncover 12 unique vulnerabilities, including 3 exclusive ones in `libtiff` (memory leak), `utf8proc` (heap buffer overflow), and `libzip` (null pointer dereference), compared to 9 vulnerabilities for OFG(∗)/CKG(∗) and 5 for the original OFG/CKG drivers. The prefix-based method detects additional vulnerabilities over the originals in `lcms`, `lunasvg`, and `utf8proc`, but misses critical defects triggered by valid option parameters, such

TABLE IV: Comparison of coverage-guided vs. random option mutation and valid vs. invalid option parameters across 10 libraries. OFG(r)/CKG(r) represent drivers with random option mutation, OFG(v)/CKG(v) represent drivers with valid options only, OFG(i)/CKG(i) represent drivers with invalid options only, while OFG(+)/CKG(+) represent the MUTATO-enhanced drivers. *CoV* denotes the average coverage across the four fuzz drivers (OFG(+/r/v/i), CKG(+/r/v/i)) under both AFL++ and LibFuzzer. *Vul* represents the total number of unique vulnerabilities discovered by these four drivers.

Library	OFG(+)/CKG(+)		OFG(r)/CKG(r)		OFG(v)/CKG(v)		OFG(i)/CKG(i)	
	CoV	*Vul*	*CoV*	*Vul*	*CoV*	*Vul*	*CoV*	*Vul*
cJSON	297	0	258(-13%)	0	278(-6%)	0	264(-11%)	0
lcms	742	1	681(-8%)	1	723(-3%)	0	623(-16%)	1
libpcap	1593	0	1549(-3%)	0	1528(-4%)	0	1287(-19%)	0
lunasvg	1432	2	1320(-8%)	2	1428(0%)	1	1145(-20%)	1
libtiff	1177	2	1017(-14%)	2	1003(-15%)	1	975(-17%)	1
utf8proc	109	3	51(-53%)	3	89(-19%)	3	47(-57%)	0
liblouis	444	1	463(4%)	1	432(-3%)	0	345(-22%)	1
xlsxio	768	2	717(-7%)	1	675(-12%)	1	596(-22%)	1
libzip	749	1	712(-5%)	0	698(-7%)	1	685(-9%)	0
libvpx	1713	0	1602(-6%)	0	1735(1%)	0	1327(-23%)	0
Count	9026	12	8371(-7%)	10	8589(-5%)	7	7294(-19%)	5

as CVE-xxx55 in libzip, due to its reliance on random prefix byte mutations. The fewer vulnerabilities found by OFG(∗)/CKG(∗) compared to OFG(+)/CKG(+) arises because the prefix-based method embeds option values within the input, limiting systematic exploration of option-driven code paths. In contrast, MUTATO separates option and input domains, enabling targeted exploration that uncovers critical vulnerabilities, particularly those triggered by valid option parameters, as seen in libzip and utf8proc.

Answer to RQ3: MUTATO outperforms the prefix-based method by treating option values and main content as separate mutation domains, achieving higher coverage and better vulnerability discovery—especially in complex formats where random prefix mutations often fail.

E. Impact of Designed Choices (RQ4)

To validate the design choices of MUTATO, we conduct an ablation study examining four key aspects that influence MUTATO's performance: (1) adaptive versus random option selection, (2) the impact of valid versus invalid option values, (3) different epsilon values, and (4) various time/execution intervals for option persistence during coverage growth.

1) Adaptive vs. Random Option Selection: To evaluate the effectiveness of MUTATO's coverage-guided adaptive option selection strategy, we compared it against a purely random approach where new option values are selected at every execution without considering coverage feedback. Table IV presents the results comparing MUTATO's adaptive strategy (OFG(+)/CKG(+)) with random option selection variants (OFG(r)/CKG(r)). The results show that the adaptive approach consistently outperforms random selection across all tested libraries. The coverage-guided option selection achieves 7% higher coverage than random selection (9,026 vs 8,371 on average) and discovers more vulnerabilities (12 vs 10). This better performance results from maintaining promising option configurations to explore newly discovered code paths, instead of switching options randomly without feedback.

TABLE V: Performance of different epsilon and option change frequency under AFL++ and LibFuzzer. *Cov* represents the average coverage of the OFG(+)/CKG(+) drivers. *Vul* denotes the total number of unique vulnerabilities discovered by both OFG(+)/CKG(+) drivers.

AFL	**Epsilon**	0.05	0.10	0.15	0.20	0.25	0.30
	Cov	10322	10212	10107	9697	9884	9865
	Vul	12	12	12	12	10	10
Libfuzzer	**Epsilon**	0.05	0.10	0.15	0.20	0.25	0.30
	Cov	7730	7712	7723	7231	7194	6959
	Vul	12	12	12	12	10	10
AFL	**Time**	1	3	5	7	10	15
	Cov	9635	9965	10322	10411	10032	9863
	Vul	10	11	12	12	12	12
Libfuzzer	**Execution**	1	100	1000	10000	100000	1000000
	Cov	7107	7023	7676	7630	7732	6942
	Vul	9	11	11	12	12	10

2) Valid vs. Invalid Option Parameters: As shown in Table IV, using only valid option parameters (OFG(v)/CKG(v)) achieves 5% less coverage than OFG(+)/CKG(+)'s valid and invalid options combined approach, while using only invalid parameters (OFG(i)/CKG(i)) results in significantly reduced coverage (19% decrease). Valid option parameters target legitimate but under-explored API configurations, revealing implementation flaws in rarely-tested code paths and discovering 7 vulnerabilities. Invalid option parameters examine input validation and error handling mechanisms, revealing boundary condition vulnerabilities and insufficient validation checks and founding 5 vulnerabilities. OFG(+)/CKG(+)'s exploration of both valid and invalid option spaces achieves comprehensive vulnerability coverage, revealing distinct vulnerability classes that single-approach methods would miss.

3) Different epsilon: To evaluate the impact of exploration probability on fuzzing performance, we conducted experiments with six different epsilon values: 0.05, 0.10, 0.15, 0.20, 0.25, and 0.30. The epsilon parameter controls the probability of exploring new option configurations even when coverage is increasing under the current option. Table V shows that when epsilon is set to 0.05, 0.10, and 0.15, MUTATO achieves similar performance in both coverage and vulnerability discovery. For AFL++, these three epsilon values maintain consistently high coverage (10,107-10,322) and detect all 12 vulnerabilities. The performance only begins to degrade when epsilon increases beyond 0.15. Similarly for LibFuzzer, epsilon values of 0.05-0.15 yield comparable results (7,712-7,730 coverage, 12 vulnerabilities). This demonstrates that the fuzzing effectiveness remains stable across this range of epsilon values. Therefore, the default epsilon = 0.05 in MUTATO is reasonable. This setting ensures sufficient exploration while avoiding unnecessary option switches, maintaining stable fuzzing performance.

4) Option Change Frequency: The frequency of option changes during coverage growth affects fuzzing efficiency. We evaluated AFL++ with time intervals of 1s, 3s, 5s, 7s, 10s, and 15s, and LibFuzzer with execution counts of 1, 100, 1,000, 10,000, 100,000, and 1,000,000, as shown in Table V. For AFL++, intervals of 5s–10s yield comparable high coverage (10,032–10,411) and detect all 12 vulnerabilities, outperforming shorter (1s: 9,635) or longer (15s: 9,863) intervals. For LibFuzzer, execution counts of 1,000–100,000 achieve stable coverage (7,630–7,732) and 11–12 vulnerabilities, with 10,000 executions balancing exploration and efficiency. Lower counts (1–100) reduce coverage (7,023–7,107), and 1,000,000 executions decrease performance (6,942). Therefore, setting AFL++ to switch options every 5s and LibFuzzer to switch after 10,000 executions is reasonable, as these values ensure effective exploration and timely detection of stagnation while maintaining high fuzzing performance.

Answer to RQ4: The ablation studies validate that MUTATO's design choices - the adaptive option selection strategy, exploration of valid and invalid options, the designed epsilon value, and option change frequency - collectively form an effective approach for option-aware fuzzing.

VI. Discussion

Fuzz Drivers. The quality of baseline fuzz drivers is a critical factor in the evaluation. We utilized automatic fuzz driver generation tools such as CKGFuzzer and OSS-Fuzz-Gen, which represent the state-of-the-art in this domain. However, despite their advanced capabilities, the quality of the generated drivers is not always optimal. We observed issues such as compilation failures and API misuse in many of the generated drivers. This variability in driver quality explains the significant performance differences between CKGFuzzer and OSS-Fuzz-Gen across certain libraries in our evaluation. It's important to note that our tool enhances existing drivers rather than generating them from scratch. Nevertheless, with higher quality baseline fuzz drivers, the improvements offered by MUTATO could potentially be even more pronounced.

OSS-Fuzz. We identified 4 vulnerabilities in APIs that have already been used in manual-written fuzz drivers within OSS-Fuzz projects, specifically in `libzip`, `lcms`, `liblouis`, and `utf8proc` as shown in Table VII (Appendix), and achieved an overall coverage increase of 12% across all tested libraries. Despite these projects being continuously fuzzed for an average of 18,060 hours by OSS-Fuzz, these vulnerabilities remained undetected. The fact that MUTATO was able to discover these 4 vulnerabilities in just 24 hours demonstrates the effectiveness of our approach in exploring API option combinations that traditional fuzzing approaches miss, even when the vulnerable APIs are directly targeted in existing fuzz drivers. This highlights the importance of systematically exploring API configuration options during fuzzing, rather than relying solely on input mutation with default or manually selected API options.

Caching of Seed-Option Pairs. MUTATO employs a stateless option selection strategy, where the fuzz driver does not maintain associations between interesting seeds and options. We explored cached seed-option pairs to restore options for interesting inputs built on manual-written drivers. Table VII shows that the caching implementation (MUTATO-Cache), while achieving only 2,006 executions per second compared to MUTATO's 13,471 executions per second (approximately 6× slower), delivers comparable performance on average and even outperforms MUTATO on some targets in terms of code coverage. This result demonstrates that the caching mechanism can be highly effective despite the significant reduction in execution throughput. The performance bottleneck in MUTATO-Cache stems from querying coverage on every execution to detect coverage gains. Therefore, we adopted interval-based queries (based on execution time intervals or execution count intervals) as an initial optimization in MUTATO to reduce costs while maintaining effectiveness. However, we acknowledge that with proper optimization — such as more efficient coverage tracking mechanisms or selective caching strategies — the caching approach could potentially outperform our stateless method. In future work, we plan to investigate optimized caching mechanisms that could leverage the potential of seed-option associations, for example by exploring lightweight coverage tracking, intelligent cache management strategies, and alternative fuzzing metrics to guide parameter changes without significantly affecting fuzzing efficiency.

VII. Related Work

Library API Fuzzing. Several research efforts have focused on automating library API fuzzing to enhance vulnerability detection. GraphFuzz [35] builds lifetime-aware dataflow graphs to create precise inputs for stateful API fuzzing. Hopper [13] employs an interpretative fuzzing approach, treating libraries as domain-specific languages to generate valid API call sequences. FUDGE [12] focuses on scalable approaches to fuzzing library APIs. NEXZZER [36] continuously infers and evolves API relations to generate diverse yet correct API sequences for effective library fuzzing. FRIES [37] targets Rust library interactions, using ecosystem-guided target generation to model API dependencies and improve fuzzing efficiency. RPG [38] enhances Rust library fuzzing with pool-based target generation and generic support, automating the creation of diverse fuzzing targets. DFUZZ [39] leverages large language models to infer edge cases for comprehensive fuzzing of deep learning library APIs. These approaches have made significant contributions to library API fuzzing but generally do not focus specifically on systematically exploring API option parameters.

Fuzz Driver Generation. Numerous research efforts have focused on automatic fuzz driver generation to address the challenge of creating effective fuzzing harnesses. APICraft [40] generates fuzz drivers for closed-source SDK libraries by analyzing API usage patterns from execution traces. [41] employs control-flow-sensitive techniques to create more effective drivers. CKGFuzzer [4] leverages code knowledge graphs to

understand API relationships for better driver generation. FuzzGen [42] constructs an Abstract API Dependence Graph from consumer code to synthesize fuzz drivers for library APIs. IntelliGen [43] automatically synthesizes drivers for fuzz testing. PromptFuzz [11] utilizes prompt engineering techniques to guide driver generation. UTOPIA [44] generates fuzz drivers by leveraging existing unit tests, while Winnie [45] focuses on Windows applications with harness synthesis. Instead of synthesizing new drivers, our approach augments existing ones by systematically mutating API option parameters, making it complementary to prior fuzz driver generation tools.

Option Exploration. Several research efforts have focused on exploring program options during fuzzing to improve bug detection. Syzkaller [46] uses API templates to fuzz kernel system calls with parameter constraints, but its kernel-specific approach requires deep fuzzer integration, unlike MUTATO's lightweight and driver-level solution. ConfigFuzz [10] proposed a systematic approach for fuzzing program configurations, targeting command-line options and configuration files. CarpetFuzz [8] automatically extracts program option constraints from documentation to guide fuzzing. CrFuzz [47] focuses on fuzzing multi-purpose programs through input validation mechanisms. POWER [6] introduces a program option-aware fuzzer that enhances bug detection ability by systematically exploring option combinations. More recently, ProphetFuzz [9] leverages large language models to predict and fuzz high-risk option combinations using only documentation, while ZigZagFuzz [7] interleaves the fuzzing of program options and files to improve effectiveness. While these approaches have made significant contributions to option-based fuzzing, they primarily focus on program-level options (command-line arguments and configuration settings) rather than API parameter options, which is the focus of our work.

VIII. Conclusion

In this paper, we presented MUTATO, a novel fuzz driver enhancement methodology that enables systematic exploration of option parameters in library API fuzzing. MUTATO implements an adaptive option parameter mutation strategy directly within the fuzz driver, remaining fuzzer-agnostic while expanding API behavior coverage. Our coverage-guided, epsilon-greedy approach intelligently balances exploration and exploitation based on coverage feedback, effectively addressing the limitations of current fuzzing techniques that neglect option parameters. The evaluation across 10 widely-used C/C++ libraries demonstrates that MUTATO-enhanced drivers achieve 14% and 13% higher code coverage with AFL++ and LibFuzzer respectively, while discovering 12 previously unknown vulnerabilities with 3 CVE assignments. These results confirm that systematic option exploration significantly improves vulnerability detection without requiring modifications to the underlying fuzzing infrastructure.

Acknowledgment

This work was partially supported by a Discovery Early Career Researcher Award Fellowship (DE250100192) awarded by the Australian Research Council (ARC) to Yuekang Li.

References

[1] W. Gao, V.-T. Pham, D. Liu, O. Chang, T. Murray, and B. I. Rubinstein, "Beyond the coverage plateau: A comprehensive study of fuzz blockers (registered report)," in *Proceedings of the 2nd International Fuzzing Workshop*, 2023, pp. 47–55.

[2] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "{AFL++}: Combining incremental steps of fuzzing research," in *14th USENIX workshop on offensive technologies (WOOT 20)*, 2020.

[3] LLVM Project, "libfuzzer – a library for coverage-guided fuzz testing," https://llvm.org/docs/LibFuzzer.html, accessed: 2025-04-21.

[4] H. Xu, W. Ma, T. Zhou, Y. Zhao, K. Chen, Q. Hu, Y. Liu, and H. Wang, "Ckgfuzzer: Llm-based fuzz driver generation enhanced by code knowledge graph," in *Proceedings of the 2025 IEEE/ACM 47th International Conference on Software Engineering: Companion Proceedings*, 2025.

[5] D. Liu, O. Chang, J. Metzman, M. Sablotny, and M. Maruseac, "Oss-fuzz-gen: Automated fuzz target generation," https://github.com/google/oss-fuzz-gen, 2024, version 1.0 [Computer software].

[6] A. Lee, I. Ariq, Y. Kim, and M. Kim, "Power: Program option-aware fuzzer for high bug detection ability," in *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2022, pp. 220–231.

[7] A. Lee, Y. Choi, S. Hong, Y. Kim, K. Cho, and M. Kim, "Zigzagfuzz: Interleaved fuzzing of program options and files," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 2, pp. 1–31, 2025.

[8] D. Wang, Y. Li, Z. Zhang, and K. Chen, "{CarpetFuzz}: Automatic program option constraint extraction from documentation for fuzzing," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 1919–1936.

[9] D. Wang, G. Zhou, L. Chen, D. Li, and Y. Miao, "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*, 2024, pp. 735–749.

[10] Z. Zhang, G. Klees, E. Wang, M. Hicks, and S. Wei, "Fuzzing configurations of program options," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 2, pp. 1–21, 2023.

[11] Y. Lyu, Y. Xie, P. Chen, and H. Chen, "Prompt fuzzing for fuzz driver generation," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 3793–3807.

[12] D. Babić, S. Bucur, Y. Chen, F. Ivančić, T. King, M. Kusano, C. Lemieux, L. Szekeres, and W. Wang, "Fudge: fuzz driver generation at scale," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 975–985.

[13] P. Chen, Y. Xie, Y. Lyu, Y. Wang, and H. Chen, "Hopper: Interpretative fuzzing for libraries," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 1600–1614.

[14] T. Yue, P. Wang, Y. Tang, E. Wang, B. Yu, K. Lu, and X. Zhou, "{EcoFuzz}: Adaptive {Energy-Saving} greybox fuzzing as a variant of the adversarial {Multi-Armed} bandit," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2307–2324.

[15] S. Luo, A. Herrera, P. Quirk, M. Chase, D. C. Ranasinghe, and S. S. Kanhere, "Make out like a (multi-armed) bandit: Improving the odds of fuzzer seed scheduling with t-scheduler," in *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, 2024, pp. 1463–1479.

[16] X. Li, X. Liu, L. Chen, R. Prajapati, and D. Wu, "Fuzzboost: Reinforcement compiler fuzzing," in *International Conference on Information and Communications Security*. Springer, 2022, pp. 359–375.

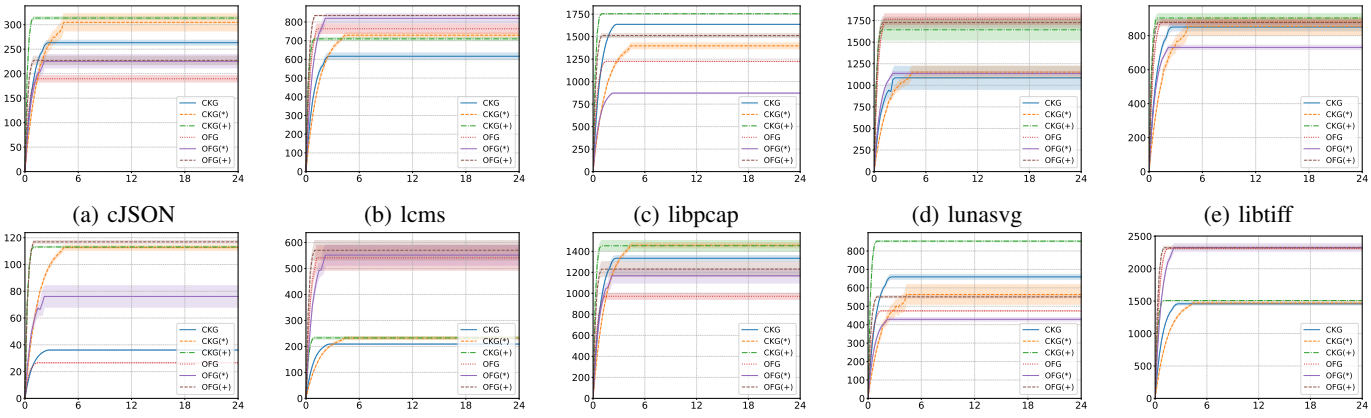
[17] X. Wang, C. Hu, R. Ma, D. Tian, and J. He, "Cmfuzz: context-aware adaptive mutation for fuzzers," *Empirical Software Engineering*, vol. 26, pp. 1–34, 2021.

[18] V. Gohil, R. Kande, C. Chen, A.-R. Sadeghi, and J. Rajendran, "Mabfuzz: Multi-armed bandit algorithms for fuzzing processors," in *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2024, pp. 1–6.

[19] C. Lemieux and K. Sen, "Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*, 2018, pp. 475–485.

[20] H. Chen, Y. Li, B. Chen, Y. Xue, and Y. Liu, "Fot: A versatile, configurable, extensible fuzzing framework," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 867–870.
[21] "cjson_read_fuzzer.c." [Online]. Available: https://github.com/DaveGamble/cJSON/blob/master/fuzzing/cjson_read_fuzzer.c
[22] "cjson." [Online]. Available: https://github.com/DaveGamble/cJSON
[23] "lcms." [Online]. Available: https://github.com/mm2/Little-CMS
[24] "liblouis." [Online]. Available: https://github.com/liblouis/liblouis
[25] "libpcap." [Online]. Available: https://github.com/the-tcpdump-group/libpcap
[26] "libtiff." [Online]. Available: https://gitlab.com/libtiff
[27] "libvpx." [Online]. Available: https://chromium.googlesource.com/webm/libvpx
[28] "libzip." [Online]. Available: https://github.com/nih-at/libzip
[29] "lunasvg." [Online]. Available: https://github.com/sammycage/lunasvg
[30] "utf8proc." [Online]. Available: https://github.com/JuliaStrings/utf8proc
[31] "xlsxio." [Online]. Available: https://github.com/brechtsanders/xlsxio
[32] A. Arya, O. Chang, J. Metzman, K. Serebryany, and D. Liu, "OSS-Fuzz." [Online]. Available: https://github.com/google/oss-fuzz
[33] "Tree-sitter: An incremental parsing system for programming tools," https://tree-sitter.github.io.
[34] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, 2018, pp. 2123–2138.
[35] H. Green and T. Avgerinos, "Graphfuzz: Library api fuzzing with lifetime-aware dataflow graphs," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1070–1081.
[36] J. Lin, Q. Zhang, J. Li, C. Sun, H. Zhou, C. Luo, and C. Qian, "Automatic library fuzzing through api relation evolvement."
[37] X. Yin, Y. Feng, Q. Shi, Z. Liu, H. Liu, and B. Xu, "Fries: Fuzzing rust library interactions via efficient ecosystem-guided target generation," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 1137–1148.
[38] Z. Xu, B. Wu, C. Wen, B. Zhang, S. Qin, and M. He, "Rpg: Rust library fuzzing with pool-based fuzz target generation and generic support," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.
[39] K. Zhang, S. Wang, J. Han, X. Zhu, X. Li, S. Wang, and S. Wen, "Your fix is my exploit: Enabling comprehensive dl library api fuzzing with large language models," *arXiv preprint arXiv:2501.04312*, 2025.
[40] C. Zhang, X. Lin, Y. Li, Y. Xue, J. Xie, H. Chen, X. Ying, J. Wang, and Y. Liu, "{APICraft}: Fuzz driver generation for closed-source {SDK} libraries," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 2811–2828.
[41] C. Zhang, Y. Li, H. Zhou, X. Zhang, Y. Zheng, X. Zhan, X. Xie, X. Luo, X. Li, Y. Liu *et al.*, "Automata-guided control-flow-sensitive fuzz driver generation." in *USENIX Security Symposium*, 2023, pp. 2867–2884.
[42] K. Ispoglou, D. Austin, V. Mohan, and M. Payer, "{FuzzGen}: Automatic fuzzer generation," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2271–2287.
[43] M. Zhang, J. Liu, F. Ma, H. Zhang, and Y. Jiang, "Intelligen: Automatic driver synthesis for fuzz testing," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2021, pp. 318–327.
[44] B. Jeong, J. Jang, H. Yi, J. Moon, J. Kim, I. Jeon, T. Kim, W. Shim, and Y. H. Hwang, "Utopia: Automatic generation of fuzz driver using unit tests," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 2676–2692.
[45] J. Jung, S. Tong, H. Hu, J. Lim, Y. Jin, and T. Kim, "Winnie: Fuzzing windows applications with harness synthesis and fast cloning," in *Proceedings of the 2021 Network and Distributed System Security Symposium (NDSS 2021)*, 2021.
[46] Google, "Syzkaller: An Unsupervised Coverage-Guided Kernel Fuzzer," https://github.com/google/syzkaller, 2016, accessed: July 27, 2025.
[47] S. Song, C. Song, Y. Jang, and B. Lee, "Crfuzz: Fuzzing multi-purpose programs through input validation," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 690–700.

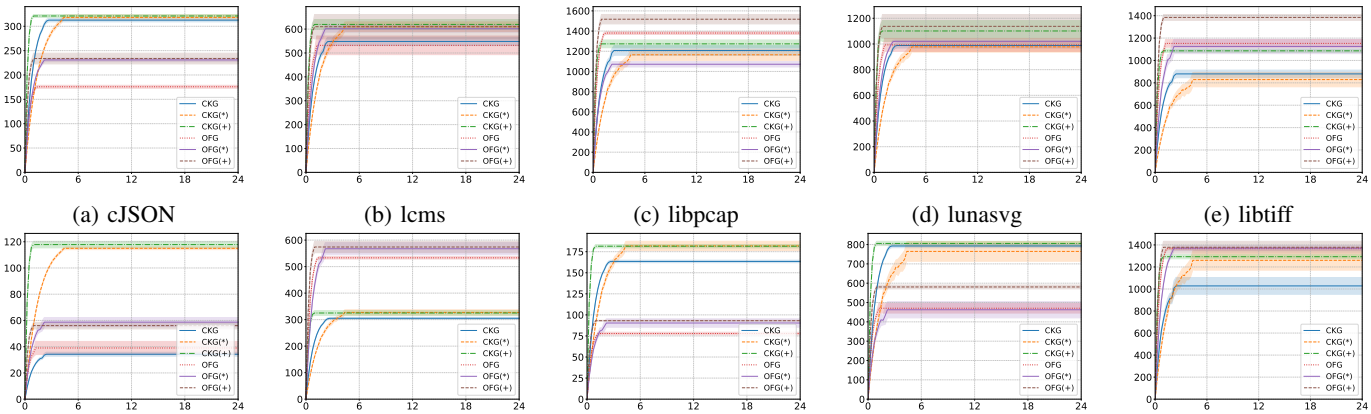
APPENDIX



(a) cJSON (b) lcms (c) libpcap (d) lunasvg (e) libtiff

(f) utf8proc (g) liblouis (h) xlsxio (i) libzip (j) libvpx

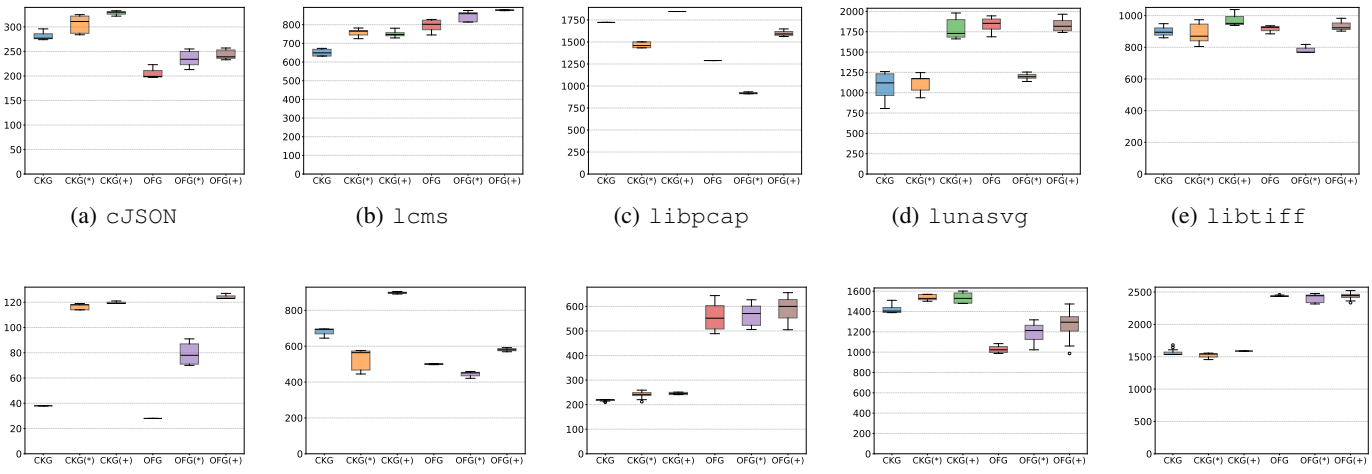
Fig. 3: Coverage Trends of CKG, OFG, CKG(∗), OFG(∗), CKG(+), and OFG(+) Under AFL++ Over 24 Hours.



(a) cJSON (b) lcms (c) libpcap (d) lunasvg (e) libtiff

(f) utf8proc (g) liblouis (h) xlsxio (i) libzip (j) libvpx

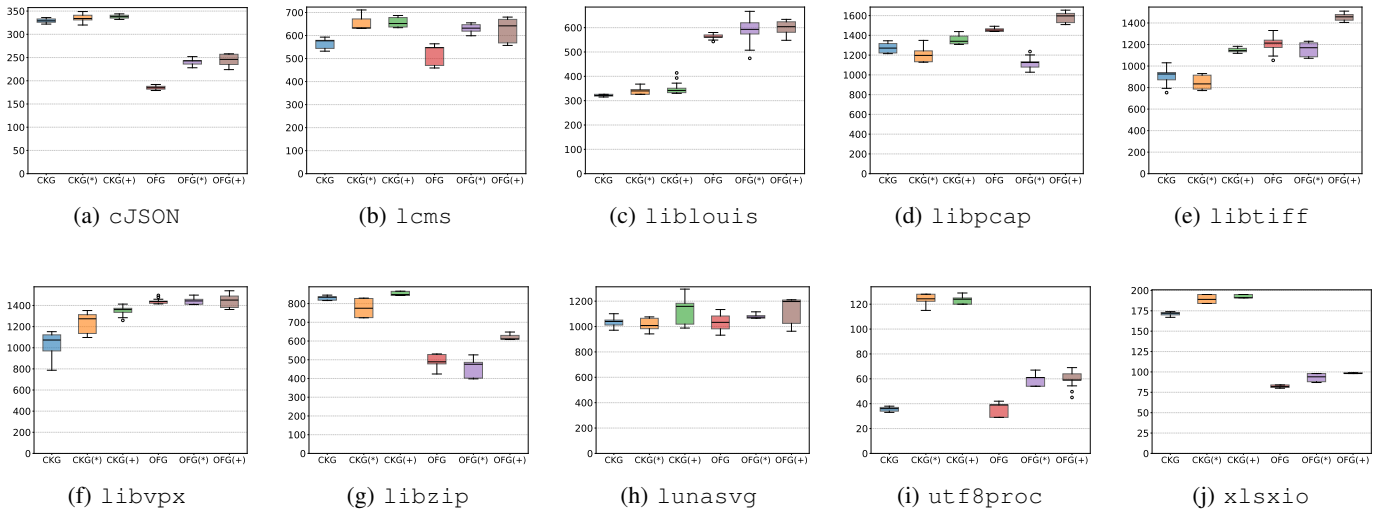
Fig. 4: Coverage Trends of CKG, OFG, CKG(∗), OFG(∗), CKG(+), and OFG(+) Under LibFuzzer Over 24 Hours.



(a) `cJSON` (b) `lcms` (c) `libpcap` (d) `lunasvg` (e) `libtiff`

(f) `utf8proc` (g) `libzip` (h) `liblouis` (i) `xlsxio` (j) `libvpx`

Fig. 5: Boxplot of Fuzzing Coverage of CKG, OFG, CKG(∗), OFG(∗), CKG(+), and OFG(+) under different modes with AFL++.



(a) cJSON (b) lcms (c) liblouis (d) libpcap (e) libtiff

(f) libvpx (g) libzip (h) lunasvg (i) utf8proc (j) xlsxio

Fig. 6: Boxplot of Fuzzing Coverage of CKG, OFG, CKG(∗), OFG(∗), CKG(+), and OFG(+) under different modes with LibFuzzer.

TABLE VI: A12 and Mann-Whitney U Test Results for Comparing OFG(+) and CKG(+) Against OFG and CKG

(a) AFL++ Fuzzer Results

Library	Metric1	Metric2	A12	p_value
cjson	OFG	OFG(+)	1	9.13e-05
cjson	CKG	CKG(+)	1	9.13e-05
lcms	OFG	OFG(+)	1	9.13e-05
lcms	CKG	CKG(+)	1	9.13e-05
libpcap	OFG	OFG(+)	1	9.13e-05
libpcap	CKG	CKG(+)	0.88	2.29e-03
lunasvg	OFG	OFG(+)	0.7	7.02e-02
lunasvg	CKG	CKG(+)	1	9.13e-05
libtiff	OFG	OFG(+)	0.89	1.81e-03
libtiff	CKG	CKG(+)	1	9.13e-05
utf8proc	OFG	OFG(+)	1	9.13e-05
utf8proc	CKG	CKG(+)	1	9.13e-05
liblouis	OFG	OFG(+)	0.71	4.06e-02
liblouis	CKG	CKG(+)	1	9.13e-05
xlsxio	OFG	OFG(+)	1	9.13e-05
xlsxio	CKG	CKG(+)	1	9.13e-05
libzip	OFG	OFG(+)	1	9.13e-05
libzip	CKG	CKG(+)	1	9.13e-05
libvpx	OFG	OFG(+)	0.84	3.96e-03
libvpx	CKG	CKG(+)	0.74	3.78e-02

(b) Libfuzzer Results

Library	Metric1	Metric2	A12	p_value
cjson	OFG	OFG(+)	1	9.13e-05
cjson	CKG	CKG(+)	0.94	5.04e-04
lcms	OFG	OFG(+)	0.99	1.23e-04
lcms	CKG	CKG(+)	1	9.13e-05
libpcap	OFG	OFG(+)	1	9.08e-05
libpcap	CKG	CKG(+)	0.86	3.64e-03
lunasvg	OFG	OFG(+)	0.86	3.64e-03
lunasvg	CKG	CKG(+)	0.83	7.01e-03
libtiff	OFG	OFG(+)	1	9.13e-05
libtiff	CKG	CKG(+)	1	9.13e-05
utf8proc	OFG	OFG(+)	1	8.98e-05
utf8proc	CKG	CKG(+)	1	9.08e-05
liblouis	OFG	OFG(+)	0.92	8.50e-04
liblouis	CKG	CKG(+)	1	4.17e-05
xlsxio	OFG	OFG(+)	1	7.69e-05
xlsxio	CKG	CKG(+)	1	8.15e-05
libzip	OFG	OFG(+)	1	9.13e-05
libzip	CKG	CKG(+)	0.98	1.64e-04
libvpx	OFG	OFG(+)	1	9.13e-05
libvpx	CKG	CKG(+)	1	9.13e-05

TABLE VII: Fuzzing statistics of manually craft(OSS-Fuzz) drivers (Manual), MUTATO-enhanced drivers (MUTATO), and MUTATO-enhanced drivers with Seed-Option Caching (MUTATO-Cache).

Metirc		**cJSON**	**lcms**	**libpcap**	**lunasvg**	**libtiff**	**utf8proc**	**liblouis**	**xlsxio**	**libzip**	**libvpx**	**Average**
Manual	*Cov*	309	1380	454	2243	3413	272	1657	82	501	2737	1305
	Vul	0	0	0	0	0	0	0	0	0	0	0
	Exec/Sec	21873	18439	9873	5274	2861	17843	14284	32183	4832	14287	14175
MUTATO	*Cov*	328	1480	501	2806	3619	319	1976	97	697	2753	1458
	Vul	0	1	0	0	0	1	1	0	1	0	4
	Exec/Sec	19872	18027	8798	4609	2726	17236	12987	31876	4814	13760	13471
MUTATO -Cache	*Cov*	328	1468	503	2712	3587	342	2007	92	684	2718	14441
	Vul	0	1	0	0	0	1	1	0	1	0	4
	Exec/Sec	3217	1987	736	527	1387	3037	729	4215	1592	2632	2006