

USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Bulbasaur: Branch-Guided Online Mutator Generation for Greybox Fuzzing

Yiyi Wang, *Tsinghua University and Zhongguancun Laboratory;* Dongsong Yu, *Zhongguancun Laboratory;* Ruiqi Dong, *Swinburne University of Technology;* Yiyang Chen, *Tsinghua University;* Xiaogang Zhu, *Adelaide University;* Chao Zhang, *Tsinghua University and Zhongguancun Laboratory*

https://www.usenix.org/conference/usenixsecurity26/presentation/wang-yiyi

This paper is included in the Proceedings of the 35th USENIX Security Symposium.

August 12–14, 2026 • Baltimore, MD, USA

ISBN 978-1-939133-58-8

Open access to the Proceedings of the 35th USENIX Security Symposium is sponsored by



جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology

ARTIFACT EVALUATED
USENIX
AVAILABLE

Bulbasaur: Branch-Guided Online Mutator Generation for Greybox Fuzzing

Yiyi Wang[1,2], Dongsong Yu[2*], Ruiqi Dong[3], Yiyang Chen[1], Xiaogang Zhu[4], Chao Zhang[1,2*]
[1]*Tsinghua University,* [2]*Zhongguancun Laboratory,*
[3]*Swinburne University of Technology,* [4]*Adelaide University*

Abstract

Coverage-guided greybox fuzzing (CGF) hinges on generating inputs that satisfy branch constraints in order to explore deep code regions. Existing CGF techniques propose various approaches to improve the probability of solving such constraints. However, they still rely on general-purpose mutators, which lack the ability to precisely manipulate relevant input bytes to satisfy complex constraints.

Our analysis reveals that LLMs can more accurately solve branch constraints by generating tailored mutators. These mutators enable the fuzzer to perform localized and reusable modifications on inputs that reach target branches, producing precise mutations to satisfy target constraints. To achieve this, we propose Bulbasaur, a branch-guided framework for online LLM-based mutator generation. Bulbasaur employs hard frontier-guided branch selection to identify critical branches, continuously collects and organizes relevant static and dynamic context to support high-quality mutator generation, and adopts an efficient and adaptive strategy to apply generated mutators during fuzzing.

We evaluate Bulbasaur on the FuzzBench and UniFuzz benchmarks. The results show that Bulbasaur improves line and branch coverage by up to 23.18% and 24.46%, respectively, over state-of-the-art fuzzers, discovers 21 vulnerabilities missed by all baselines, and identifies 15 previously unknown vulnerabilities on OSS-Fuzz, all of which have been assigned CVE identifiers.

1 Introduction

Coverage-guided greybox fuzzing (CGF) is an effective automated software testing technique that discovers vulnerabilities by using code coverage feedback to guide the exploration toward untested code paths [23, 31, 53, 56]. The efficiency of CGF primarily hinges on the ability to generate inputs to satisfy branch constraints guarding deeper code regions [8, 9, 41]. If a constraint cannot be satisfied, the code guarded by this branch remains untested, preventing potential vulnerabilities in that region from being discovered. These constraints vary widely, from simple magic number checks to complex semantic conditions that depend on the interplay between input structure, data values, and program logic [11, 13, 17, 51].

Due to the random nature of fuzzing, existing CGF techniques propose diverse solutions to improve the probability of solving branch constraints. Seed scheduling, a coarse-grained solution, improves this probability by prioritizing the seeds that have a higher chance of exploring new code regions [4, 39, 50]. A finer-grained approach is byte-level scheduling, which improves the probability by prioritizing certain bytes [2, 17, 47, 55]. However, the efficiency of these two solutions is constrained by mutators (e.g., bit flip) because they randomly select mutators that are the same across different constraints. While mutator scheduling can improve the likelihood of solving constraints, such solutions still use the same general mutators [19, 21, 30, 45, 49]. Since real-world branch constraints often involve complex combinations of multiple bytes, the general-purpose mutators lack the ability to precisely manipulate the relevant bytes to satisfy these constraints. As a result, mutators become a key bottleneck for modern fuzzing efficiency. A natural question we ask is: "can we generate tailored mutators on-the-fly to efficiently solve diverse constraints?"

With the development of powerful large language models (LLMs), it becomes possible to generate tailored mutators that significantly improve the probability of solving branch constraints. Our key insight is that LLMs can leverage their prior knowledge and code understanding capabilities to capture the underlying semantics of these constraints, and accordingly generate branch-specific mutators. These mutators perform localized and reusable modifications on existing inputs that reach the target branches. By preserving the structural properties of these inputs, they precisely operate on regions related to branch constraints, producing accurate mutations that satisfy target conditions and enable deeper code exploration.

While LLMs offer a promising direction for generating tailored mutators, realizing this vision in practice introduces

*Corresponding authors.

three key challenges:

C1: **Which branches are worth generating mutators for?** Real-world programs often contain a large number of branch constraints. Invoking LLMs to generate mutators indiscriminately for all branches would incur unacceptably high time and financial costs. Therefore, it is essential to identify the hard branches that genuinely limit coverage and are worth the cost of mutator generation.

C2: **How to generate high-quality mutators?** To accurately understand branch constraints, LLMs require much richer context than local code snippets, including both static semantic information from the program and dynamic data observed during execution. Therefore, a mechanism is needed to automatically collect and organize the required static and dynamic context, and present it to the LLM in an appropriate form, enabling the generation of high-quality mutators.

C3: **How to efficiently and adaptively use mutators?** LLM-generated mutators are executed at high frequency during fuzzing. Implementing them as scripts incurs substantial disk I/O overhead due to frequent file read/write operations, significantly reducing fuzzing throughput. Moreover, since the effectiveness of a mutator depends on how accurately the LLM understands the target branch constraint, some mutators may perform poorly in practice. Therefore, it is essential to efficiently integrate these mutators and dynamically evaluate and schedule them based on their runtime performance.

Our Approach. To address these challenges, we propose Bulbasaur, a branch-guided framework for online LLM-based mutator generation.

For C1, given the large number of branches in real-world programs, we first identify those that have not been fully covered, i.e., branches for which at least one outgoing edge remains uncovered. Such branches typically lie on the coverage frontier and act as key bottlenecks limiting further coverage. During fuzzing, we continuously monitor the coverage status of these branches. When a branch remains uncovered for an extended period, we regard it as a hard branch and invoke the LLM to generate a tailored mutator specifically for this branch, thereby avoiding unnecessary LLM invocations.

For C2, to enable the LLM to generate high-quality mutators, we provide it with both static and dynamic information. At the static level, we design a program query agent to interact with the LLM and supply branch-relevant semantic information on demand, helping the LLM to understand the semantic context of the target branch constraints. On the dynamic side, we adopt an operand-aware mutator template, which enables generated mutators to directly leverage runtime branch operands from the target branch to guide mutations when the LLM deems such operands informative.

For C3, we implement LLM-generated mutators as dynamically linked libraries and load them into the fuzzer at runtime, enabling them to be invoked directly within the fuzzing loop. This design avoids costly disk I/O operations and significantly improves execution efficiency. In addition, we design an online adaptive mutator scheduling strategy that dynamically adjusts the usage probability of mutators based on their performance during fuzzing, progressively prioritizing those that are truly effective.

We evaluate Bulbasaur on the FuzzBench [33] and UniFuzz [24] benchmarks. The results demonstrate that Bulbasaur achieves improvements of 23.18% in line coverage and 24.46% in branch coverage over the second-best baseline, AFL++ with CmpLog, on UniFuzz. In addition, Bulbasaur discovers 21 more vulnerabilities that baselines miss on UniFuzz. Furthermore, Bulbasaur identifies 15 previously unknown real-world vulnerabilities on the OSS-Fuzz platform, all of which have been assigned CVE identifiers.

Contributions. In summary, our contributions are as follows:

- We propose Bulbasaur, a branch-guided framework for online LLM-based mutator generation that addresses three key challenges: (1) selectively identifying hard branches worth targeting via hard frontier-guided selection, (2) generating high-quality mutators by providing LLMs with rich static context (via a program query agent) and dynamic operands, and (3) efficiently integrating mutators via dynamic linking and adaptively scheduling them based on runtime effectiveness.
- Based on the proposed design, we implement a prototype system of Bulbasaur. Our implementation is publicly available at `https://github.com/vul337/Bulbasaur.git`.
- We conduct extensive evaluations on FuzzBench and UniFuzz benchmarks. The results show that Bulbasaur achieves 23.18% higher line coverage and 24.46% higher branch coverage than the second-best baseline on UniFuzz, discovers 21 vulnerabilities missed by all baselines on UniFuzz, and identifies 15 CVEs on OSS-Fuzz.

2 Motivation

We use a simplified and abstracted version of the CRC verification logic for PNG chunks in `libpng` [36], as illustrated in Figure 1, to motivate our work. Specifically, Lines 12–14 compute the CRC value over the chunk type and payload, while Lines 16–20 extract the CRC stored in the input file. The target branch at Line 22 compares the computed CRC with the stored one to determine whether the current chunk is valid. Only when this constraint is satisfied will the chunk be accepted and further parsed.

2.1 Limitations of Existing Approaches

Existing fuzzing approaches struggle to solve such CRC verification branches effectively.

Taint-guided approaches [2, 17, 25] can identify the input bytes within the PNG chunk type and payload that influence the target branch. However, they still rely on general-purpose mutators, which provide little guidance on how to precisely

```
1. int parse_png_chunk(uint8_t *input, size_t size)
2. {
3.     if (size < 12) return -1;
4.     /* Layout:
5.      * [4B type][N bytes data][4B crc]
6.      */
7.     uint8_t *type = input;
8.     uint8_t *data = input + 4;
9.     uint8_t *crc_field = input + size - 4;
10.    size_t data_len = size - 8;
11.    /* Compute CRC(Type || Data) */
12.    uint32_t crc = crc32(0, Z_NULL, 0);
13.    crc = crc32(crc, type, 4);
14.    crc = crc32(crc, data, data_len);
15.    /* Read CRC from input */
16.    uint32_t stored =
17.        (crc_field[0] << 24) |
18.        (crc_field[1] << 16) |
19.        (crc_field[2] << 8)  |
20.        (crc_field[3]);
21.    /* Target branch: Integrity check */
22.    if (crc != stored) {
23.        return -1;   // reject
24.    }
25.    return 0;       /* Target edge: accept */
26. }
```

Figure 1: CRC verification branch in libpng.

```
1. pub fn mutate(
2.     buf: *mut Vec<u8>,
3.     op1: *const Vec<u8>, // runtime crc
4.     op2: *const Vec<u8>, // stored crc
5. ) -> bool {
6.     unsafe {
7.         let buf= &mut *buf;
8.         let op1 = &*op1;
9.         let op2 = &*op2;
10.        let mut matches = Vec::new();
11.        // Collect all matches
12.        for i in 0..=buf.len() - 4 {
13.            if &buf[i..i + 4] == op2 {
14.                matches.push(i);
15.            }
16.        }
17.        if matches.is_empty() {
18.            return false;
19.        }
20.        // Randomly choose one
21.        let mut rng = thread_rng();
22.        let idx = matches[rng.gen_range(0..matches.len())];
23.        // Replace
24.        buf[idx..idx + 4].copy_from_slice(op1);
25.        true
26.    }
27. }
```

Figure 2: Mutator for the CRC verification branch in libpng.

manipulate these bytes to generate a valid CRC value that satisfies the underlying constraint.

Hybrid concolic approaches like Driller [41] and QSYM [51] combine fuzzing with symbolic execution to solve constraints. However, they struggle with branches involving complex library functions: symbolically modeling CRC32 produces formulas that cause solver timeouts, while treating it as opaque prevents reasoning.

Grammar-based approaches [3, 14, 35, 44] rely on manually specified input specifications. While these specifications describe the syntactic structure of PNG files, they primarily focus on format-level constraints rather than fine-grained semantic dependencies among fields, such as CRC checksums. Consequently, they fail to directly construct consistent CRC values that satisfy integrity verification.

2.2 Towards LLM-Generated Mutators

In contrast to prior approaches, we explore the use of LLMs to generate tailored mutators for satisfying specific branch constraints.

Specifically, we provide the LLM with the `libpng` project description and the source-level context relevant to the target branch, including the corresponding outgoing control-flow edge. In addition to the static information often used by existing approaches, we observe that dynamic execution data, especially branch operands, can provide crucial runtime guidance for mutation. For the CRC verification branch, operands expose both the computed CRC value (`op1`) and the stored value (`op2`), enabling the mutator to directly repair mismatches without understanding the full CRC computation algorithm. Therefore, we design a mutator template that allows generated mutators to access both branch operands. During mutator generation, we encourage the LLM to leverage these operands in the generated mutators when they are deemed useful.

We then require the LLM to generate a tailored mutator that mutates inputs reaching the target branch at Line 22 so as to traverse the target edge at Line 25.

As illustrated in Figure 2, after understanding the semantics of the CRC verification branch, the LLM generates a tailored mutator. Specifically, it scans the input for byte sequences that match `op2` (i.e., the stored CRC value), which correspond to candidate CRC fields in the PNG chunk (Lines 12–16). When multiple such matches exist, it randomly selects one candidate position to modify (Lines 21–22). It then overwrites the selected position with `op1` (i.e., the runtime-computed CRC value), thereby repairing the checksum and satisfying the integrity verification constraint (Line 24).

2.3 Why LLM-Generated Mutators over Direct Input Generation?

Recent LLM-assisted constraint solving approaches, such as ConcoLLMic [29], attempt to satisfy branch conditions by directly generating the entire input. However, we observe that such direct-generation approaches inherently suffer from the path condition accumulation problem, which significantly limits their effectiveness, as detailed below.

The Path Condition Accumulation Problem. To solve a target branch, an LLM-generated input must not only solve the target constraint but also simultaneously preserve all preceding path conditions. As the target branch gets deeper, these path conditions accumulate. Generating an input that satisfies the target without violating any earlier checks becomes highly challenging.

Empirical Study. To validate this, we conduct an empirical study using the open-source implementation of ConcoLLMic powered by DeepSeek v3.2. We select one shallow branch and one deep branch (defined by a threshold of 20

Table 1: Solving success rates over 100 attempts for shallow and deep branches: ConcoLLMic (full-input generation) vs. our method (local rewriting).

Target Project	ConcoLLMic		Our Method	
	Shallow	Deep	Shallow	Deep
mujs	**81%**	59%	75%	**86%**
exiv2	76%	30%	**100%**	**94%**
objdump	0%	0%	**92%**	**45%**

preceding path conditions) from `mujs`, `exiv2`, and `objdump` in the UniFuzz benchmark. The specific branch locations are `jsregexp.c:17` (shallow) and `json.c:275` (deep) for `mujs`; `webpimage.cpp:518` (shallow) and `jp2image.cpp:348` (deep) for `exiv2`; and `i386-dis.c:14017` (shallow) and `i386-dis.c:17585` (deep) for `objdump`. For each branch, we supply ConcoLLMic with an input that reaches the target branch but fails to satisfy it, allowing the LLM to collect the necessary path conditions. We then query ConcoLLMic to solve the target branch 100 times and calculate the success rates.

As shown in Table 1, while ConcoLLMic achieves reasonable success rates on the shallow branches of `mujs` and `exiv2`, its performance drops sharply on deeper branches. Furthermore, in `objdump`, the complexity of preceding binary format checks causes the success rate to drop to 0% for both shallow and deep branches.

Local Rewriting vs. Full-Input Generation. We argue that once a seed already satisfies the preceding conditions, the LLM should not expend excessive reasoning capacity on prefix path conditions. Instead, it only needs to understand the semantics of the target branch to generate a lightweight mutator that locates the relevant bytes within the input and performs localized modifications. This reduces the task from full-input generation to local input rewriting, thereby circumventing the path condition accumulation problem.

To demonstrate this, we use our method to generate mutators for the same six branches, applying them to the corresponding reaching inputs to satisfy the target constraints. Across 100 generation attempts per branch, our method achieves the success rates shown in Table 1. Notably, our method significantly outperforms ConcoLLMic on the deep branches of `exiv2` and `mujs`, and successfully resolves the complex binary checks in `objdump` where direct generation fails entirely.

3 Design

3.1 Overview

In this section, we present the overall design of Bulbasaur. Bulbasaur leverages LLMs to generate branch-specific mutators, thereby helping to solve hard branch constraints in programs and reach deeper code regions. Figure 3 illustrates its overall workflow.

During fuzzing, Bulbasaur first executes a seed and collects dynamic branch operands from the branches it traverses (①). In the mutation stage, it employs an adaptive mutator scheduling strategy to select a subset of branches and applies the corresponding branch-specific mutators to mutate the seed, generating new test cases (②). It then executes the generated test cases and collects runtime feedback (③). Based on the feedback, Bulbasaur determines whether the applied mutators are effective and dynamically adjusts the selection probabilities of the corresponding branches, enabling it to prioritize those that are truly effective (④).

Meanwhile, the feedback is also used to update the branch states maintained in the branch database, including detecting newly reached branches and determining whether certain branches have been fully covered (i.e., all of their outgoing edges have been covered) (⑤). Based on this online monitoring, Bulbasaur employs hard frontier-guided branch selection to identify hard branches and submits them to the LLM for further processing (⑥).

On the LLM side, Bulbasaur designs a program query agent to equip the LLM with program exploration capabilities. During an initial, one-time setup phase, the agent supplies the necessary program context, enabling the LLM to generate a high-level description of the target program (⑦). During fuzzing, guided by this description and the branch-specific context provided by the program query agent, the LLM generates tailored mutators for hard branches submitted by the branch database. These mutators follow an operand-aware mutator template, allowing them to directly leverage runtime branch operands (⑧). The generated mutators are dynamically loaded into Bulbasaur's mutator database and subsequently applied during the mutation stage (⑨).

In the following sections, we present our solutions to the three challenges introduced in Section 1, respectively.

3.2 Hard Frontier-Guided Branch Selection

Real-world programs typically contain a large number of branches. Invoking LLMs to generate tailored mutators for all branches, either before or during fuzzing, would incur unacceptably high time and financial costs. In practice, many branches do not require LLM-generated mutators, as they are either not yet reached during fuzzing or can be easily covered using random mutation strategies (e.g., the havoc strategy [52]).

To address this challenge, Bulbasaur introduces a hard frontier-guided branch selection strategy, which dynamically and selectively identifies valuable branches during fuzzing and triggers the LLM to generate mutators only when necessary.

Specifically, Bulbasaur maintains a branch database to track newly reached branches and determine their coverage sta-

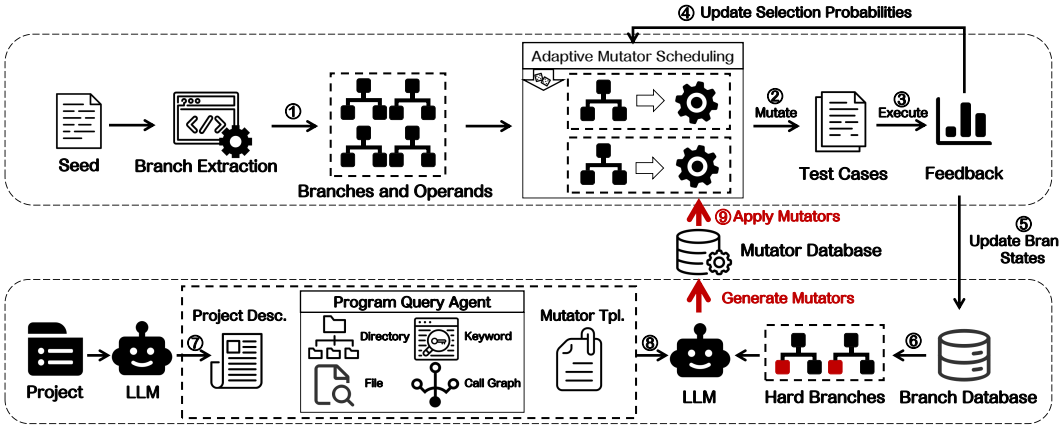


Figure 3: The workflow of Bulbasaur.

tus based on execution feedback. It first identifies frontier branches whose outgoing edges are not fully covered. Among these frontier branches, Bulbasaur further selects those that have remained uncovered for a predefined period as hard branches, indicating persistent difficulty under existing mutation strategies. Only such hard frontier branches are selected for LLM-based mutator generation.

By applying this strategy, Bulbasaur filters out branches that are either unreachable or already fully covered, and concentrates the limited LLM budget on a small set of hard frontier branches that genuinely limit coverage.

In our evaluation, we empirically set the predefined period to 6 hours, striking a balance between generating sufficient mutators and controlling LLM invocation costs.

3.3 Branch-Specific Mutator Preparation

To generate high-quality mutators, LLMs require much richer context than local code snippets alone. Bulbasaur provides the LLM with three complementary types of information: (1) a **project description** that summarizes the program's input formats and processing logic, generated once before fuzzing via LLM-based project exploration; (2) **branch-specific static context** containing source-level semantics relevant to the target branch, retrieved on demand through multi-round interaction with the program query agent; and (3) an **operand-aware mutator template** that exposes runtime branch operands, enabling generated mutators to localize the relevant input region precisely. Together, these inputs allow the LLM to understand what a target branch is checking, where in the input the relevant bytes reside, and how to rewrite them to satisfy the constraint.

3.3.1 Project Description

Before fuzzing begins, Bulbasaur performs a one-time initialization in which the LLM, guided by the program query agent, explores the target project's documentation and source code to produce a high-level **project description**. This description summarizes the program's overall purpose, supported input formats and data structures, important constants and magic values, and key parsing and validation logic. Rather than re-exploring the project for every hard branch, this description is prepended to all subsequent mutator generation prompts, giving the LLM a stable global understanding of the target program. Appendix A.1 shows the prompt template used during this phase.

3.3.2 Program Query Agent

To provide the LLM with richer branch-related static semantic information, Bulbasaur introduces a program query agent that enables the LLM to interactively explore the program and retrieve relevant information on demand. Specifically, the agent equips the LLM with the following four program exploration capabilities:

Directory Queries. The agent can inspect the project's directory structure to locate relevant documentation and source files.

File Queries. The agent can access documentation and source code snippets with specified line ranges and configurable context widths.

Keyword Search. The agent can perform keyword-based searches over specified files and return the matched lines along with their corresponding locations.

Call Graph Queries. Prior to fuzzing, the call graph of the target program is extracted using SVF [42], augmented with Andersen-style pointer analysis [1] to help resolve indirect-call targets. The agent can query caller–callee relationships between functions and their corresponding source locations.

During mutator generation, the LLM engages in a multi-round interaction with the program query agent. Starting from the branch and edge source context provided in the generation prompt, the LLM may issue additional queries—e.g., reading

```
pub fn mutate(
    buf: *mut Vec<u8>,
    op1: *const Vec<u8>,
    op2: *const Vec<u8>,
) -> bool {
// ... (mutation logic omitted)
}
```

Figure 4: Operand-aware mutator template.

related parsing functions, searching for relevant constants, or tracing caller chains—to gather the information it needs. Once it has collected sufficient context, the LLM proceeds to formulate the required mutation logic.

3.3.3 Operand-Aware Mutator Template

Dynamic branch operands, i.e., the concrete runtime values involved in branches, provide richer and more direct guidance for mutation. In many cases, operands collected from a target branch can serve as value-level mutation sources or as anchors for localizing mutation sites.

To enable LLM-generated mutators to leverage such branch operands during mutation, Bulbasaur introduces an operand-aware mutator template. When generating mutators, the LLM follows the template illustrated in Figure 4. In addition to the input buffer, this template explicitly takes the two runtime operands of the target branch as parameters. During prompt construction, Bulbasaur explicitly informs the LLM that, when branch operands are deemed helpful for satisfying the target branch constraint, they may be incorporated into the mutation logic to guide mutation.

During fuzzing, Bulbasaur executes the seed, records dynamic branch operands along the traversed branches, and feeds them to the corresponding LLM-generated mutators during mutation.

3.3.4 Mutator Generation

Given the project description, branch-specific static context, and the operand-aware mutator template, the LLM generates a targeted mutator for the hard branch. Specifically, the LLM (i) infers what condition the target branch is checking and why existing inputs fail to satisfy it; (ii) identifies the most relevant input region by reasoning over the input format, branch semantics, and runtime operands; and (iii) rewrites that region in a semantics-aware, structure-preserving manner to satisfy the branch constraint. **The generated mutator is best understood as a lightweight, branch-specific parser/rewriter**: rather than applying random byte flips, it encodes knowledge of the input format and the branch's semantic requirement, and applies targeted modifications to inputs that already reach the branch.

The complete prompt template orchestrating this generation process is provided in Appendix A.2.

3.3.5 Regeneration and Unbreakable-Branch Detection

Despite rich static and dynamic context, LLM-generated mutators may still fail to solve certain branch constraints, primarily because some constraints are inherently complex, while others are input-independent and cannot be influenced by mutations. To address these issues, Bulbasaur introduces two additional mechanisms.

Mutator Regeneration. If an LLM-generated mutator fails to solve the target branch constraint within a predefined time window, Bulbasaur re-invokes the LLM to generate a new mutator, leveraging both the previously generated mutator and the branch-specific context obtained by the program query agent. The newly generated mutator directly replaces the previous one and is used for subsequent fuzzing. To balance effectiveness and cost, we set the time window to 6 hours, and perform at most one regeneration attempt for each branch constraint.

Unbreakable-Branch Detection. During mutator generation, the prompt first instructs the LLM to assess whether the target branch is input-dependent. If the LLM determines that the branch cannot be influenced by input mutations, it returns a special marker, and Bulbasaur subsequently skips mutator generation for this branch to avoid ineffective attempts.

3.4 Efficient and Adaptive Mutator Execution

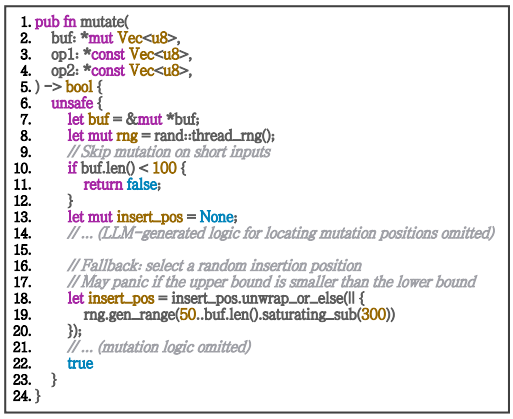
After the LLM generates tailored mutators, several issues arise in their practical application during fuzzing. On the one hand, these mutators are invoked frequently, and script-based implementations can introduce significant disk I/O overhead. On the other hand, their effectiveness depends on how accurately the LLM captures the underlying branch constraints, and some generated mutators may be ineffective in practice.

To address these issues, Bulbasaur introduces an efficient and adaptive mutator execution mechanism that enables lightweight and performance-aware integration of LLM-generated mutators into the main fuzzing loop.

3.4.1 Dynamically Linked Mutators

Many existing LLM-enhanced fuzzing approaches [15, 54] generate inputs using Python scripts. This design is generally acceptable for one-shot input generation, where each script is invoked once to produce a small number of specific inputs. However, in fuzzing, mutators are executed at very high frequency. As a result, if mutators are implemented as scripts, the fuzzer must repeatedly write seeds to disk and read generated test cases back. This introduces substantial disk I/O overhead and significantly degrades overall fuzzing throughput.

To mitigate this overhead, Bulbasaur compiles LLM-generated mutators into Rust dynamic libraries (`cdylib`) and loads them into the fuzzer at runtime. This design enables efficient in-memory mutation with minimal I/O overhead, while providing flexible management of tailored mutators.



```
1. pub fn mutate(
2.     buf: *mut Vec<u8>,
3.     op1: *const Vec<u8>,
4.     op2: *const Vec<u8>,
5. ) -> bool {
6.     unsafe {
7.         let buf = &mut *buf;
8.         let mut rng = rand::thread_rng();
9.         // Skip mutation on short inputs
10.        if buf.len() < 100 {
11.            return false;
12.        }
13.        let mut insert_pos = None;
14.        // ... (LLM-generated logic for locating mutation positions omitted)
15.
16.        // Fallback: select a random insertion position
17.        // May panic if the upper bound is smaller than the lower bound
18.        let insert_pos = insert_pos.unwrap_or_else(|| {
19.            rng.gen_range(50..buf.len().saturating_sub(300))
20.        });
21.        // ... (mutation logic omitted)
22.        true
23.    }
24. }
```

Figure 5: An unsafe LLM-generated mutator.

During fuzzing, Bulbasaur automatically compiles newly generated mutators into dynamic libraries. If the compilation of a mutator fails, Bulbasaur captures the corresponding compiler error messages and feeds them back to the LLM, enabling automatic repair of faulty mutators.

3.4.2 Isolated Mutator Execution

Although loading mutators directly into the fuzzer can significantly reduce mutation overhead, LLM-generated mutators may contain logical errors. Executing such mutators within the main fuzzing process can therefore introduce several failure modes:

Fuzzer crashes: logical bugs or unchecked boundary conditions in mutators may trigger illegal behavior (e.g., invalid memory accesses), causing the main fuzzer to crash.

Fuzzer hangs: mutators may contain infinite loops or excessively complex logic, causing the fuzzing process to stall indefinitely.

Resource exhaustion: mutators may repeatedly allocate memory, leading to uncontrolled memory growth and eventual out-of-memory failures.

Figure 5 illustrates an example of an LLM-generated mutator that contains a subtle bug. Because the check `buf.len() < 100` at Line 10 is insufficient, the expression `buf.len().saturating_sub(300)` at Line 19 may evaluate to a value smaller than 50 (e.g., when the buffer length is exactly 100, the result becomes 0). As a consequence, `gen_range()` may receive an invalid range in which the upper bound is smaller than the lower bound, causing a runtime panic. If such a mutator were executed directly within the main fuzzer process, it could terminate the entire fuzzing campaign.

To solve this problem, Bulbasaur adopts a fork-based isolated mutator execution mechanism. Specifically, whenever the main fuzzer invokes an LLM-generated mutator, it first forks a mutation worker subprocess. The mutator is executed exclusively within this isolated subprocess, and the mutated input is returned to the main process via shared memory.

This mechanism enables Bulbasaur to safely leverage LLM-generated mutators without compromising the stability of the main fuzzer. Specifically, crashes in the mutator are detected by inspecting the subprocess's exit status. To handle hangs, Bulbasaur enforces a timeout of 2 seconds and forcibly terminates the subprocess if it fails to return within this time limit. In addition, to prevent resource exhaustion, Bulbasaur restricts the memory usage of each mutation worker to 64 MB, ensuring that excessive resource consumption does not lead to out-of-memory failures. Meanwhile, the use of shared memory ensures that mutator execution remains efficient.

Algorithm 1: Match-Replace Mutator

Input: buf: input buffer; op_1, op_2: branch operands
Output: buf': mutated buffer

```
if br.op1 = ∅ and br.op2 = ∅ then
    return buf
r ← RandomChoice({1,2}) ;
if r = 1 then
    src ← br.op1 ;
    dst ← br.op2 ;
else
    src ← br.op2 ;
    dst ← br.op1 ;
P ← FindAllOccurrences(buf, src) ;
if P = ∅ then
    return buf
p ← RandomChoice(P) ;
buf' ← ReplaceAt(buf, p, src, dst) ;
return buf'
```

3.4.3 Match-Replace as a First-Line Mutator

During fuzzing, Bulbasaur first executes seeds and collects dynamic branch operands from the branches it traverses. Before invoking the LLM to generate tailored mutators for hard branches, Bulbasaur applies a lightweight match-replace mutator that leverages these operands to solve simple input-dependent constraints, such as magic-number checks.

The match-replace mutator is a dictionary-style mutator. As illustrated in Algorithm 1, for a given target branch, it randomly selects one operand (`op1` or `op2`) as the match source, denoted as `src`, and uses the other operand as the replacement target, denoted as `dst`. It then scans the input buffer to locate all occurrences of `src` and records their offsets in a candidate set $\mathcal{P}$. If $\mathcal{P}$ is non-empty, Bulbasaur randomly selects one position from the set and replaces the corresponding `src` with `dst`.

Our evaluation results in Section 5.6 confirm that employing match-replace as a first-line mutator yields noticeable coverage improvements across many targets.

3.4.4 Adaptive Mutator Scheduling

Since each seed typically traverses a large number of branches, it is associated with a large pool of corresponding mutators. As a result, many candidate mutators are available for each seed, and the effectiveness of mutation strongly depends on selecting appropriate target branches and applying their corresponding mutators.

To address this issue, Bulbasaur introduces an online adaptive mutator scheduling strategy that prioritizes mutators during fuzzing. Specifically, Bulbasaur probabilistically selects branches according to their current selection probabilities, applies the corresponding mutators to the current seed, and updates these probabilities based on the execution feedback of the generated test cases.

Probabilistic Modeling. We model the process of branch selection as a Bernoulli process to capture the uncertainty in mutation effectiveness. For the i-th branch, let θ_i denote the probability that applying its associated mutator yields a successful outcome. Let P_i be a Bernoulli random variable indicating whether a mutation attempt on branch i is successful. We assume:

$$P_i \sim \text{Bernoulli}(\theta_i), \quad \theta_i \sim \text{Beta}(\alpha_i, \beta_i), \tag{1}$$

where α_i and β_i represent the prior counts of successful and failed mutation outcomes associated with branch i, respectively.

Each time the mutator corresponding to branch i is applied, we observe a binary outcome indicating success or failure. Let s and f denote the numbers of observed successes and failures. Using Bayesian updating, the posterior distribution of θ_i is given by:

$$\theta_i \mid D \sim \text{Beta}(\alpha_i + s,\ \beta_i + f), \tag{2}$$

where $D = (s, f)$ denotes the observed mutation outcomes.

Branch Selection Strategy. To make selection decisions, Bulbasaur adopts Thompson Sampling [7, 43] to balance exploration and exploitation. At each scheduling step, a value is sampled from the Beta posterior of each branch and used as its selection weight, with branches associated with higher sampled values being more likely to be selected.

Reward Definition. The branch-specific mutators in Bulbasaur are designed to be reusable. Rather than merely satisfying a branch constraint once, they aim to continuously guide diverse inputs toward deep code regions guarded by the target branch and facilitate the exploration of varied execution paths within these regions.

To quantify how effectively a mutator contributes to this goal, Bulbasaur defines a reward signal based on coverage improvement in under-explored regions.

Specifically, Bulbasaur maintains hit counts for all edges and defines the set E_{rare} as the union of (i) the 10% least frequently exercised edges and (ii) edges that have not yet been covered. A mutation is considered successful if the resulting test case exercises at least one edge in E_{rare}. This reward design encourages Bulbasaur to prioritize mutators that are more effective in exploring under-explored code regions.

Algorithm 2: Adaptive Mutator Scheduling

Input: *seed*: current seed to mutate

```
1  B ← ExtractOperands(seed) ;
2  foreach br ∈ B do
3      br_weight[br] ← BetaSample(br_beta[br]);
4  E_rare ← GetRareEdge(edge_hitcnt) ;
5  for i ← 1 to N_trials do
6      buf ← Clone(seed);
7      k ← 1 + RandBelow(K_branches);
8      applied ← ∅;
9      brs ← WeightedPick(br_weight, k);
10     foreach br ∈ brs do
11         mut ← mutator_database[br];
12         if ¬mut(buf, br) then
13             br_beta[br].β ← br_beta[br].β + 1;
14         else
15             applied ← applied ∪ {br};
16     if applied.empty() then
17         continue;
18     (reward, edges) ← Execute(buf, E_rare) ;
19     foreach br ∈ applied do
20         if reward = 1 then
21             br_beta[br].α ← br_beta[br].α + 1;
22         else
23             br_beta[br].β ← br_beta[br].β + 1;
24     foreach edge ∈ edges do
25         edge_hitcnt[edge] ← edge_hitcnt[edge] + 1;
```

Algorithm Workflow. As illustrated in Algorithm 2, during each mutation stage, Bulbasaur first executes the seed to collect the set of branch operands (Line 1). For these branches, Bulbasaur samples selection weights from their respective Beta distributions (Lines 2-3).

Bulbasaur then performs N_{trials} mutation trials. In each trial, it randomly selects k branches according to the sampled weights, where k is randomly drawn from the range $[1, K_{\text{branches}}]$ (Lines 5-9). For each selected branch, Bulbasaur applies its associated mutator to the seed (Line 11). If multiple operand pairs are observed for a branch, Bulbasaur randomly selects one and passes it to the corresponding mutator.

If a mutator fails to modify the input, it is treated as a failure, and the Beta distribution of the corresponding branch is updated accordingly (Lines 12-13). If the mutation succeeds, the generated test case is executed, and the Beta distributions of all applied mutators are updated based on whether any rare edges are exercised (Lines 18-23). Meanwhile, edge hit frequencies are updated to maintain an up-to-date E_{rare} set (Lines 24-25).

We empirically set N_{trials} to 128 to ensure that each seed generates a sufficient number of test cases in the mutation

stage. We set K_{branches} to 16 to involve a limited subset of branches in each mutation trial, increasing mutation diversity while preventing excessive modifications that could lead to destructive mutations. In addition, when a new LLM-generated mutator is loaded, Bulbasaur resets the corresponding branch's Beta distribution to enable re-evaluation of its effectiveness.

4 Implementation

Based on the above design, we implement Bulbasaur. The core fuzzer of Bulbasaur is implemented in Rust and consists of approximately 8K lines of code. For seed selection and power scheduling, Bulbasaur follows the exploration mode strategy of AFL++ [16].

During mutation, in addition to the proposed branch-specific mutation strategy, Bulbasaur retains the havoc mutation strategy from AFL++ to enable broad random exploration.

For instrumentation, Bulbasaur extends the AFL++ 4.31c instrumentation tool with approximately 6K lines of additional C++ code.

4.1 Non-blocking LLM Invocation

Many existing LLM-enhanced fuzzing approaches [15, 32, 48, 54] invoke LLMs directly in the main fuzzing loop, which blocks fuzzing execution due to LLM latency and significantly degrades overall throughput.

To address this limitation, Bulbasaur adopts a multi-threaded architecture that decouples LLM invocation from the main fuzzing loop. The main thread continuously executes test cases and updates branch states, while a dedicated LLM thread asynchronously invokes the LLM to generate mutators on demand. This design prevents LLM latency from blocking fuzzing and preserves high execution throughput.

4.2 Branch-Based Instrumentation

Bulbasaur augments the AFL++ LLVM-based instrumentation pass and introduces two complementary instrumentation modes.

Feedback Instrumentation. The feedback instrumentation operates in a lightweight mode, with runtime overhead comparable to that of native AFL++ instrumentation. During instrumentation, it identifies branches by tracking CMP instructions and establishes an explicit mapping between branches and their outgoing edges. By monitoring edge coverage at runtime, Bulbasaur can determine whether a branch has been reached and whether all outgoing edges have been fully covered.

Operand Instrumentation. The operand instrumentation records dynamic branch operands observed during program execution. For each branch, it records up to 32 operand pairs, corresponding to the first 32 executions of that branch. To identify string-comparison branches, the operand instrumentation analyzes whether a CMP instruction directly depends on the return value of a string comparison function. For such string-comparison branches, Bulbasaur uses the arguments of the corresponding string comparison function as branch operands, enabling more effective mutation guidance.

In addition, during instrumentation, Bulbasaur relies on debug information to obtain source-code locations for each branch, its outgoing edges, and all functions, in order to support the program query agent.

5 Evaluation

We evaluate Bulbasaur on two benchmarks, FuzzBench [33] and UniFuzz [24], to answer the following research questions:

- **RQ1:** How effective is Bulbasaur in improving code coverage?
- **RQ2:** How effective is Bulbasaur in discovering vulnerabilities?
- **RQ3:** How many branches trigger LLM-based mutator generation in Bulbasaur, and what is the associated LLM cost?
- **RQ4:** What is the runtime overhead introduced by Bulbasaur?
- **RQ5:** How does each component contribute to the performance of Bulbasaur?

We compare Bulbasaur against the widely used fuzzer AFL++ [16] and its CmpLog variant (i.e., AFL++ running in the default CmpLog-enabled mode, which incorporates taint tracking from RedQueen [2]), as well as four recent representative coverage-guided fuzzing approaches—Fox [40], NestFuzz [12], ShapFuzz [55], and ZTaint [47].

We further compare Bulbasaur with G2FUZZ [54], an LLM-enhanced fuzzer that leverages LLMs to synthesize and mutate input generators for non-textual input formats. This comparison is presented separately in Section 5.3.

In all experiments, we use DeepSeek v3.2 [26] for LLM-based components and conduct evaluations on a machine running Ubuntu 20.04 with 128 CPU cores and 1.8 TB of memory (Intel Xeon Platinum 8358). On FuzzBench, we use FuzzBench at commit `2a2ca6` with AFL++ at commit `56d5aa`, following the benchmark's default specifications. On UniFuzz, we use AFL++ 4.31c. All fuzzers run with their default configurations in single-core pinned containers, with one container per trial. Each experiment is repeated for 10 independent trials to ensure statistical reliability.

5.1 Coverage Evaluation

On FuzzBench, we compare Bulbasaur with CmpLog, which represents the default high-performance configuration of AFL++. Other baselines are not fully supported on FuzzBench

Table 2: Comparison of code coverage for Bulbasaur and CmpLog across 22 targets on FuzzBench. R_{Cov} denotes the relative improvement.

Target	Bulbasaur	CmpLog		Bulbasaur-5d	CmpLog-5d	
	Coverage	Coverage	R_{Cov}	Coverage	Coverage	R_{Cov}
bloaty	**6412**	6231	2.91%	**6821**	6741	1.19%
curl	**7694**	6812	12.94%	**7909**	7027	12.55%
freetype2	**12487**	11412	9.42%	**14087**	12264	14.87%
harfbuzz	**11134**	10897	2.18%	**11321**	11118	1.83%
jsoncpp	**525**	520	0.96%	**525**	520	0.96%
lcms	**2267**	1987	14.09%	**2282**	2241	1.83%
libjpeg	**2549**	2547	0.08%	**2551**	2548	0.12%
libpcap	**3143**	2854	10.13%	**3173**	2871	10.52%
libpng	**2024**	2003	1.05%	**2029**	2013	0.79%
libxml2	**16008**	15731	1.76%	**16143**	15968	1.10%
libxslt	**11379**	11218	1.44%	**11601**	11238	3.23%
mbedtls	**3961**	2748	44.14%	**4001**	2849	40.47%
openh264	**9556**	9555*	0.06%	**9659**	9651*	0.08%
openssl	**5830**	5823*	0.17%	**5836**	5834*	0.03%
openthread	**3752**	3079	21.86%	**4387**	3118	40.70%
proj4	**8080**	6988	15.73%	**8768**	7847	11.74%
re2	2868	**2877**	-0.31%	2869	**2882**	-0.45%
sqlite3	18024	**20258**	-11.03%	**20523**	20476*	0.23%
systemd	**241**	236	2.12%	**242**	236	2.55%
vorbis	**1331**	1261	5.55%	**1345**	1278	5.24%
woff2	**1222**	1181	3.47%	**1223**	1204	1.58%
zlib	**465**	460	1.09%	**470**	463	1.51%
Sum	**130952**	126678	3.37%	**137765**	130387	5.66%

*Not statistically significant (p>0.05, Mann-Whitney U, two-sided).

(e.g., Fox integration remains unsuccessful[1]); therefore, we use UniFuzz for a broader comparison involving all six baselines.

UniFuzz integrates a diverse set of real-world programs, many of which contain previously reported vulnerabilities, making it suitable for evaluating both coverage and vulnerability discovery.

For coverage measurement, FuzzBench uses Sanitizer Coverage [37]. On UniFuzz, we measure line and branch coverage by replaying the final corpus of each fuzzer on targets instrumented with Sanitizer Coverage.

In our FuzzBench experiments, we selected 22 target programs, each running for 24 hours. The code coverage results for Bulbasaur and CmpLog are presented in Table 2. Overall, Bulbasaur achieves 3.37% higher coverage than CmpLog. Among the 22 targets, Bulbasaur outperforms CmpLog on 20 of them, with coverage improvements exceeding 10% on six targets. The improvement is statistically significant (p<0.05) on 18 of those 20 targets.

Among the two remaining targets on which Bulbasaur performs worse than CmpLog, `re2` and `sqlite3`, the coverage degradation on `re2` is marginal (0.31%), whereas `sqlite3` exhibits a more noticeable drop (11.03%). Therefore, we focus our subsequent analysis on `sqlite3`. Further analysis reveals that Bulbasaur executes substantially fewer runs than CmpLog within 24 hours (74.3M vs. 395.7M).

We attribute this difference to two main factors. First, Bulbasaur incurs additional overhead from collecting dynamic branch operands. Second, many LLM-generated mutators involve more complex mutation logic, resulting in higher mutation overhead. Together, these factors reduce the overall fuzzing throughput of Bulbasaur, which in turn leads to lower coverage on `sqlite3` compared with CmpLog.

Furthermore, we conducted a long-term fuzzing experiment in which each fuzzer was executed continuously for five days. The results are reported in Table 2 under the columns labeled Bulbasaur-5d and CmpLog-5d. Bulbasaur-5d outperforms CmpLog-5d on 21 out of 22 targets. Notably, on `sqlite3`, Bulbasaur-5d achieves higher coverage than CmpLog-5d. Overall, Bulbasaur-5d attains 5.66% higher coverage than CmpLog-5d. These results indicate that, despite the runtime overhead introduced by Bulbasaur, it converges to a higher coverage upper bound in long-term fuzzing.

In addition, the evaluation on UniFuzz further confirms the strong performance of Bulbasaur over the six baseline fuzzers. As shown in Table 3, Bulbasaur achieves the highest line coverage and branch coverage on 17 out of 19 target programs. Notably, on 16 of these 17 targets, the improvements achieved by Bulbasaur are statistically significant compared to all evaluated baselines. Overall, Bulbasaur outperforms the second-best fuzzer, CmpLog, by 23.18% in line coverage and 24.46% in branch coverage.

For the two targets on which Bulbasaur performs relatively poorly, namely `sqlite3` and `tcpdump`, we attribute the performance degradation to the following reasons. On `sqlite3`, Bulbasaur still exhibits relatively low execution throughput, as the total number of executions achieved by Bulbasaur is approximately half of that achieved by AFL++. This limited throughput leads to inferior coverage performance on this target. On `tcpdump`, Bulbasaur achieves lower coverage than Fox, although it still outperforms AFL++ and CmpLog. We attribute Fox's advantage to its branch-distance guided strategy, which explicitly tracks branch distances and attempts to satisfy constraints by minimizing numeric distance.

5.2 Vulnerability Discovery Evaluation

We collect statistics on the number of unique vulnerabilities discovered by each fuzzer on UniFuzz. Vulnerability uniqueness is determined by hashing the top three functions from the ASAN output, following widely adopted crash deduplication practices [22]. Among the 19 target programs, vulnerabilities were discovered in 16 of them. We report the mean number of unique vulnerabilities per trial across 10 trials in Table 4. Among these 16 targets, Bulbasaur achieves the highest average on 13 of them, and for 9 of these 13 targets, the improvement over all evaluated baselines is statistically significant. Across all 16 targets, Bulbasaur discovers 49.8 more unique vulnerabilities than the second-ranked ShapFuzz, indicating a clear advantage.

To further characterize the uniqueness of Bulbasaur's vulnerability discovery capability, we plot an UpSet diagram to illustrate the overlap and diversity of vulnerabilities discov-

[1] https://github.com/google/fuzzbench/pull/2051

Table 3: Comparison of code coverage for different fuzzers across 19 targets on UniFuzz.

Target	Bulbasaur		CmpLog		AFL++		Fox		NestFuzz		ShapFuzz		ZTaint	
	L.	B.	L.	B.	L.	B.	L.	B.	L.	B.	L.	B.	L.	B.
cflow	**3136**	**1584**	3135	1582	3135	1582	3133	1580	3134	1576	3135	1583	3135	1582
exiv2	**10724**	**4334**	7198	2603	5987	1967	6488	2194	5432	1723	6012	1997	6491	2154
flvmeta	**936**	**421**	**936***	**421***	933	420	**936***	**421***	932	419	**936***	**421***	**936***	**421***
ffmpeg	**74723**	**35310**	51628	24048	47802	22265	36847	17604	34623	16763	40228	19263	39124	18668
imginfo	**9087**	**4128**	5601	2548	5602	2474	5531	2456	5412	2376	5554	2469	6412	2855
infotocap	**3250**	**1750**	3194	1682	2891	1515	2902	1505	3072	1581	3115	1645	3043	1601
jhead	**1338**	**850**	987	601	531	279	757	451	487	263	701	389	758	452
jq	**5497**	**2165**	5472	2147	5478	2148	5474	2145	-	-	5471	2147	5476	2151
gdk	**9052**	**3663**	7791	3109	7756	3078	6656	2647	3694	1416	7793	3112	7799	3101
lame	**8350**	**2880**	8232	2832	8176	2806	8219	2828	8183	2816	8191	2819	8213	2825
mujs	**7985**	**4561**	5831	3421	5556	3228	5731	3359	5524	3201	5614	3294	5724	3360
nm	**12104**	**5059**	8382	3590	8428	3651	8665	3740	7326	3118	8492	3638	7694	3319
objdump	**16828**	**8056**	13289	6496	12389	6183	10993	5303	10558	5156	12178	6123	12788	6343
pdftotext	**19127**	**8031**	15689	6568	16289	6867	14607	6088	13824	5748	15297	6453	14047	5843
sqlite3	42876	16817	42918*	16931*	**43718**	**17351**	40992	15831	41213	15982	42812*	16824*	42525*	16763*
tcpdump	29974	17023	27185	15397	18864	10258	**32048**	**18484**	29728*	16434*	30152*	16994*	30363*	17102*
tiffsplit	**5264**	**2652**	4472	2184	4289	2138	4479	2221	4483	2216	4808	2349	4442	2207
mp3gain	**3467**	**1167**	3238	1069	3078	1028	3293	1098	3135	1057	3186	1063	3156	1053
mp42aac	**6108**	**2474**	3879	1538	2618	993	2642	1026	2712	1053	2943	1161	2698	1044
Sum	**269826**	**122925**	219057	98767	203520	90231	200393	90981	183472	82898	206618	93744	204824	92844

NestFuzz fails to start on the `jq` target. ShapFuzz encounters runtime errors on `ffmpeg`, `nm`, and `objdump`, leading to premature termination.
*Not statistically significant (p>0.05, Mann-Whitney U, two-sided).

Table 4: Unique vulnerabilities discovered across 16 targets on UniFuzz.

Target	BB	CL	AFL	FOX	NF	SF	ZT
cflow	1.2	**3.0**	**3.0**	2.6	**3.0**	**3.0**	2.1
exiv2	**22.4**	12.6	7.1	6.9	14.7	5.2	7.8
flvmeta	**2.0**	**2.0***	**2.0***	**2.0***	**2.0***	**2.0***	**2.0***
ffmpeg	**1.0**	**1.0***	0.6	0	0	0	0
imginfo	**1.0**	0	0	0	0	0	0
infotocap	**3.1**	2.0	1.0	2.0	2.0	3.0*	2.0
jhead	**23.2**	9.7	3.0	5.8	2.1	6.9	5.0
gdk	**19.8**	7.6	6.9	11.8	0	15.7	14.8
lame	**5.1**	4.0	4.0	4.0	4.0	5.0*	4.0
mujs	**1.0**	0	0	0	0	0	0
nm	0	0*	0*	**0.7**	0*	0*	0*
objdump	**7.3**	6.9	4.9	2.1	1.0	3.9	4.0
pdftotext	10.2	5.1	**17.9**	5.9	5.0	17.8	7.9
tiffsplit	**9.4**	3.0	2.1	3.9	6.7	4.9	3.8
mp3gain	**12.3**	7.8	6.9	7.5	7.9	8.0	7.0
mp42aac	**6.2**	2.9	0	1.0	0	0	0
Sum	**125.2**	67.6	59.4	56.2	48.4	75.4	60.4

BB: Bulbasaur; **CL:** CmpLog; **AFL:** AFL++; **FOX:** Fox; **NF:** NestFuzz; **SF:** ShapFuzz; **ZT:** ZTaint.
*Not statistically significant (p>0.05, Mann-Whitney U, two-sided).

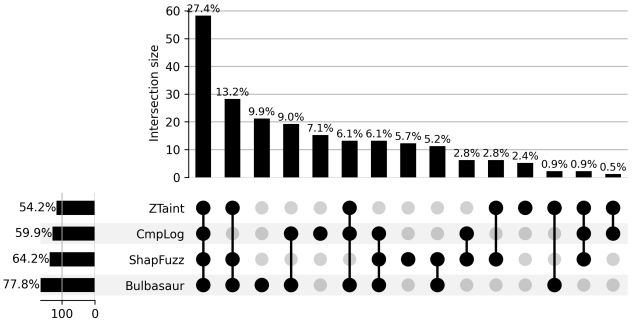


Figure 6: UpSet plot illustrating the orthogonality and diversity of unique vulnerabilities discovered by different fuzzers.

ered by different fuzzers. We compare the top four fuzzers, as shown in Figure 6. The upper bar chart shows the number and percentage of intersections among vulnerability sets, while the left bar chart presents the number of unique vulnerabilities discovered by each fuzzer and their proportion of the total.

Overall, the four fuzzers collectively discover 212 unique vulnerabilities. Among them, Bulbasaur identifies 165 (77.8% of the total) vulnerabilities, including 21 (9.9%) that were exclusively discovered by Bulbasaur. In comparison, ShapFuzz, CmpLog, and ZTaint identify 136 (64.2%), 127 (59.9%), and 115 (54.2%) vulnerabilities, respectively. These results show that Bulbasaur identifies many vulnerabilities missed by other fuzzers.

Note that the UpSet diagram is constructed using the union of all trials, whereas Table 4 reports averages across trials, which explains the discrepancy between the two datasets.

5.3 Comparison with G2FUZZ

In addition, we compare Bulbasaur with G2FUZZ, an LLM-enhanced fuzzer that leverages LLMs to synthesize and mutate input generators, which are then used to enrich seed features and enhance input diversity. We evaluate both approaches in terms of code coverage and the number of unique vulnerabilities discovered on UniFuzz.

It is important to note that G2FUZZ relies on manually specified input format definitions to guide the LLM in generator synthesis and mutation. The official G2FUZZ project provides such definitions for only 10 targets on UniFuzz, while the remaining targets are not supported. For example,

Table 5: Comparison of code coverage and unique vulnerabilities (Vul.) for Bulbasaur-G2FUZZ, Bulbasaur, and G2FUZZ on UniFuzz.

Target	Bulbasaur-G2FUZZ			Bulbasaur			G2FUZZ		
	L.	B.	Vul.	L.	B.	Vul.	L.	B.	Vul.
exiv2	**10984**	**4429**	**30.7**	10724	4334	22.4	6712	2218	15.8
flvmeta	**936**	**421**	**2.0**	**936***	**421***	**2.0***	**936***	**421***	**2.0***
ffmpeg	**79247**	**38021**	**1.0**	74723	35310	**1.0***	62018	29347	0.0
imginfo	**9468**	**4286**	**1.0**	9087	4128	**1.0***	6831	3062	0.0
jhead	1328	840	**23.9**	**1338***	**850***	23.2*	1196	736	10.7
gdk	**9194**	**3729**	17.8	9052	3663	**19.8**	8047	3214	11.8
pdftotext	**20583**	**8708**	**10.3**	19127	8031	10.2*	16934	6958	6.8
tiffsplit	5216	2641	9.3	**5264***	**2652***	**9.4***	4718	2374	7.9
mp3gain	**3467**	**1167**	11.7	**3467***	**1167***	**12.3***	3278	1094	5.3
mp42aac	**6212**	**2538**	**8.2**	6108	2474	6.2	4784	1847	2.1
Sum	**146635**	**66780**	**115.9**	139826	63030	107.5	115454	51271	62.4

*Not statistically significant (p>0.05, Mann-Whitney U, two-sided). Superscript on Bulbasaur values: not significantly different from Bulbasaur-G2FUZZ; on G2FUZZ values: not significantly different from Bulbasaur.

programs such as `nm` and `objdump` take ELF binaries as input, for which there is currently a lack of mature Python libraries capable of generating high-quality ELF files. In addition, for `mujs` and `sqlite3`, the inputs are inherently difficult to abstract into generator-friendly formats.

As shown in Table 5, across the 10 supported targets, Bulbasaur achieves the highest code coverage and discovers the most unique vulnerabilities compared with G2FUZZ. Notably, these improvements are statistically significant on all targets except `flvmeta`. Specifically, it attains 21.11% higher overall line coverage, 22.93% higher overall branch coverage, and identifies 45.1 more unique vulnerabilities than G2FUZZ. These results indicate that, compared with generator-based approaches that remain agnostic to branch constraints, Bulbasaur can explicitly perceive branch states and focus on solving critical bottleneck branches, leading to higher coverage and more vulnerability discoveries.

More importantly, our approach is largely orthogonal to existing LLM-based input generation techniques, such as G2FUZZ. Since Bulbasaur only modifies the mutation and feedback collection components, it can seamlessly incorporate seeds generated by G2FUZZ.

To demonstrate this orthogonality, we further implement a variant named Bulbasaur-G2FUZZ, which integrates G2FUZZ into Bulbasaur via an additional thread. At the initial stage of fuzzing, this thread uses LLM-synthesized generators to produce an extra batch of seeds. During fuzzing, it periodically monitors progress. If no new paths appear within a predefined window, it invokes G2FUZZ to mutate generators and produce additional seeds.

The results show that Bulbasaur-G2FUZZ achieves the highest coverage on 8 out of the 10 evaluated targets. On the remaining two targets, namely `jhead` and `tiffsplit`, its coverage is slightly lower than that of Bulbasaur, with a modest gap of around 1%.

We attribute the advantage of Bulbasaur-G2FUZZ to the complementary strengths of the two approaches. G2FUZZ is effective at enriching the initial seeds with previously unseen high-level features, while Bulbasaur excels at performing fine-grained, branch-specific mutations on newly introduced features. For example, our analysis on `ffmpeg` shows that Bulbasaur-G2FUZZ explores substantially more H.264 CABAC decoding logic. This improvement mainly results from G2FUZZ generating MP4 seeds containing CABAC-enabled H.264 streams, which are absent from the initial corpus. Building on these seeds, Bulbasaur further refines the associated bitstream features, enabling deeper exploration.

Overall, these results indicate that Bulbasaur can effectively cooperate with LLM-based input generation techniques that focus on enriching high-level input features. While such generators introduce missing semantic features, Bulbasaur complements them with fine-grained, branch-specific mutations that enable deeper exploration.

In addition, Appendix B compares the LLM costs of Bulbasaur, G2FUZZ, and Bulbasaur-G2FUZZ.

5.4 Branch Selection Analysis

We further analyze the total number of branches (TB), the number of branches that trigger LLM-based mutator generation, which we refer to as LLM branches (LB), the ratio of LLM branches to total branches (LR), and the LLM API cost per target across 19 UniFuzz targets.

As shown in Table 6, all targets contain a substantial number of branches. However, Bulbasaur selects only a small subset of hard branches for mutator generation. Across all 19 targets, the proportion of LLM branches remains below 15%, with a global ratio of only 1.82%. Moreover, the average LLM cost per selected branch is less than $0.02.

These results indicate that Bulbasaur effectively focuses LLM resources on critical branches, maintains a well-controlled cost per branch, and achieves strong performance while keeping the overall API cost within a reasonable range.

We note that the branches counted here refer to the branches instrumented by Bulbasaur, which differ from the branch coverage metrics reported in Table 3 using Sanitizer Coverage. Sanitizer Coverage uses a different instrumentation and counting scheme, often resulting in larger branch counts than our branch database. This difference in measurement does not affect our conclusions but is clarified here to avoid confusion.

In addition, we provide a detailed analysis of how many LLM branches are eventually solved by the generated mutators in Appendix C.

5.5 Overhead Analysis

Bulbasaur introduces three primary sources of overhead: LLM calls, mutator compilation, and fork-based isolation.

LLM Calls. As described in Section 3.4, LLM calls for mutator generation run in a separate background thread and do

Table 6: Branch selection statistics and LLM cost.

Program	TB	LB	LR	Cost
cflow	1957	183	9.35%	$5.36
exiv2	14017	349	2.49%	$5.98
flvmeta	1726	19	1.10%	$0.54
ffmpeg	151918	361	0.24%	$6.29
imginfo	6834	355	5.19%	$6.49
infotocap	3876	109	2.81%	$2.47
jhead	743	24	3.23%	$0.67
jq	2807	337	12.01%	$6.73
gdk	2284	234	10.25%	$5.79
lame	3879	386	9.95%	$7.12
mujs	7868	430	5.47%	$6.44
nm	14361	278	1.94%	$5.68
objdump	21531	267	1.24%	$6.48
pdftotext	13166	344	2.61%	$6.71
sqlite3	15214	445	2.92%	$8.56
tcpdump	15656	478	3.05%	$7.47
tiffsplit	3964	222	5.60%	$5.67
mp3gain	920	95	10.33%	$2.50
mp42aac	5017	311	6.20%	$5.48
Sum	287738	5227	1.82%	$102.43

TB: Total Branches; **LB:** LLM Branches; **LR:** LLM Branch Ratio; **Cost:** Total LLM API cost (USD).

not block the main fuzzing loop.

Mutator Compilation. Generated mutators are compiled into shared libraries before being loaded. Even for the target with the most LLM branches (`tcpdump`), fewer than 500 branches require mutator compilation across the entire fuzzing campaign (see Table 6), resulting in negligible cumulative overhead.

Fork-Based Isolation. As detailed in Section 3.4.2, Bulbasaur utilizes fork-based isolation to protect the main fuzzer process from potentially unsafe mutators. To measure the overhead of this fork operation, we implement **B-NoFork**, a no-fork variant of Bulbasaur that eliminates fork-based isolation by executing mutators directly in the main process. In this variant, only mutators previously validated as safe are used to ensure stability. We compare the fuzzing throughput of Bulbasaur and B-NoFork across all 19 UniFuzz targets, as reported in Table 7.

Overall, B-NoFork achieves a 3.36% higher aggregate throughput compared to Bulbasaur. While the overhead is statistically significant on 9 out of 19 targets, with the worst-case throughput reduction reaching 8.30% (on `cflow`), the majority of targets experience minimal or statistically insignificant slowdowns. This indicates that, from a holistic perspective, the fork-based isolation cost remains modest.

The limited overhead can be attributed to the fact that LLM-generated mutators often perform complex work, such as format-aware parsing of the entire input buffer, making mutator execution itself the dominant cost rather than the fork operation. These results confirm that fork-based isolation is a practical design choice: it provides strong safety guarantees against mutator crashes and hangs with minimal overhead.

Table 7: Fuzzing throughput (total executions in millions over 24 hours) for Bulbasaur vs. B-NoFork across 19 UniFuzz targets. R_{Thr} denotes the relative throughput difference.

Program	Bulbasaur	B-NoFork	R_{Thr}
cflow	17.35M	**18.92M**	−8.30%
exiv2	27.06M	**27.92M**	−3.08%
ffmpeg	2.35M	**2.43M***	−3.29%
flvmeta	62.46M	**63.32M***	−1.36%
gdk	12.78M	**12.88M***	−0.78%
imginfo	7.20M	**7.59M**	−5.14%
infotocap	28.72M	**31.21M**	−7.98%
jhead	83.18M	**85.97M***	−3.25%
jq	4.67M	**4.81M**	−2.91%
lame	6.15M	**6.43M**	−4.35%
mp3gain	**19.76M**	19.65*M**	0.56%
mp42aac	19.20M	**19.93M***	−3.66%
mujs	20.35M	**21.44M***	−5.08%
nm	21.66M	**21.74M***	−0.37%
objdump	10.37M	**10.57M***	−1.89%
pdftotext	8.72M	**8.89M***	−1.91%
sqlite3	16.72M	**17.08M**	−2.11%
tcpdump	25.16M	**26.61M**	−5.45%
tiffsplit	24.31M	**25.31M**	−3.95%
Sum	418.17M	**432.70M**	−3.36%

*Not statistically significant (p>0.05, Mann-Whitney U, two-sided).

5.6 Ablation Study

Bulbasaur leverages LLM-generated mutators to solve branch constraints. The strategies that directly impact Bulbasaur's performance include branch-specific mutator preparation, which exploits both static and dynamic program context to generate high-quality mutators; match-replace as a first-line mutator, which enables efficient handling of simple input-dependent constraints; and adaptive mutator scheduling, which dynamically prioritizes effective mutators. Therefore, we conducted an ablation study on these strategies to investigate their contributions to Bulbasaur's effectiveness. Furthermore, we implement a variant that completely disables LLM-based mutator generation to demonstrate the fundamental effectiveness of Bulbasaur's LLM-guided approach in solving branch constraints.

Specifically, we evaluate four variants of Bulbasaur: (1) Bulbasaur without branch-specific mutator preparation, where only local context is used for mutator generation (**B-Context**); (2) Bulbasaur without match-replace as a first-line mutator (**B-NoMR**); (3) Bulbasaur without adaptive mutator scheduling, where mutators are selected uniformly at random (**B-Static**); and (4) Bulbasaur with all LLM-based mutator generation disabled, retaining all other components (**B-NoLLM**). The experimental results are summarized in Table 8.

Across the 19 target programs, the full version of Bulbasaur achieves the highest coverage on 18 targets. Notably, on 9 of these targets, the improvements achieved by Bulbasaur are statistically significant compared to all evaluated variants.

Table 8: Ablation study of Bulbasaur variants on UniFuzz.

Target	Bulbasaur		B-Context		B-NoMR		B-Static		B-NoLLM	
	L.	B.	L.	B.	L.	B.	L.	B.	L.	B.
cflow	**3136**	**1584**	**3136***	**1584***	**3136***	**1584***	**3136***	**1584***	**3136***	**1584***
exiv2	**10724**	**4334**	8891	3421	9743	3918	10127	4103	6709	2371
flvmeta	**936**	**421**	**936***	**421***	**936***	**421***	**936***	**421***	**936***	**421***
ffmpeg	**74723**	**35310**	66987	32419	65412	31854	69712	33914	58453	28007
imginfo	**9087**	**4128**	6751	3048	5957	2683	6015	2709	5602	2485
infotocap	**3250**	**1750**	3249*	**1750***	3247*	1749*	3244*	1748*	3224	1732
jhead	**1338**	**850**	1330*	846*	1312	828	1310	834	1195	753
jq	**5497**	**2165**	5483*	2157*	5492*	2162*	5493*	2164*	5471	2149
gdk	**9052**	**3663**	8556	3434	8806	3519	8765	3517	7370	2942
lame	**8350**	**2880**	8296	2845	8341*	2879*	8343*	2879*	8239	2819
mujs	**7985**	**4561**	7451	4206	7627	4294	7853	4420	5622	3220
nm	**12104**	**5059**	11850	4962	11453	4787	12017*	5008*	8720	3564
objdump	**16828**	**8056**	16809*	8053*	16092	7659	16678	8001	13244	6509
pdftotext	**19127**	**8031**	17021	7152	16741	6950	18323	7614	16282	6831
sqlite3	42876	16817	**43180***	**17017***	42618*	16697*	42864*	16827*	42202	16441
tcpdump	**29974**	**17023**	27385	15208	24858	13746	28504	15927	21752	12625
tiffsplit	**5264**	**2652**	5112	2573	5190	2624	5148	2606	4671	2354
mp3gain	**3467**	**1167**	3464*	1164*	3456*	1163*	3465*	1165*	3135	1037
mp42aac	**6108**	**2474**	6044	2465*	5966	2449	5986	2450	4074	1689
Sum	**269826**	**122925**	251931	114725	246383	111966	257919	117891	220037	99533

*Not statistically significant (p>0.05, Mann-Whitney U, two-sided).

These results demonstrate the effectiveness of the evaluated strategies in Bulbasaur.

Additionally, we observe that while the overall coverage of B-NoLLM is comparable to that of CmpLog, their performance diverges significantly on specific targets. For instance, CmpLog achieves 24.98% higher line coverage than B-NoLLM on `tcpdump`, whereas B-NoLLM outperforms CmpLog by 13.22% on `ffmpeg`. We attribute these performance variances primarily to the underlying differences between B-NoLLM's match-replace mutation strategy and the CmpLog strategy in AFL++.

5.7 Real-world Vulnerability Discovery

To further evaluate the vulnerability discovery capability in practice, we applied Bulbasaur to the OSS-Fuzz [38] platform. OSS-Fuzz integrates multiple fuzzers (e.g., LibFuzzer [28], AFL++ [16], and Honggfuzz [18]) and provides continuous fuzzing for widely used open-source projects.

Despite the long-term fuzzing efforts already performed on OSS-Fuzz, Bulbasaur was able to discover 15 previously unknown real-world vulnerabilities across several projects, all of which have been assigned CVE identifiers (see Appendix D).

6 Limitations and Future Work

Handling Complex Semantic Constraints. Although Bulbasaur can effectively solve a portion of branch constraints and improve code coverage across many programs, some complex branches in real-world software remain challenging. For example, certain branches depend on state-machine flags or high-level semantic states that are constructed through multi-stage and intricate input parsing. Without understanding the complete state-construction process, the LLM may fail to accurately capture the semantics of such branches and thus cannot generate effective mutators.

As future work, Bulbasaur could be combined with additional static analysis techniques (e.g., AST-based analysis or control/data-flow analysis) to provide richer semantic information. Meanwhile, effective mechanisms are required to filter, organize, and de-duplicate such information, so that only concise and relevant context is presented to the LLM, avoiding information overload and potential performance degradation.

Robust Source-Level Mapping. Currently, Bulbasaur's program query agent relies on debug info to map instrumented code to source locations. In real-world binaries, some basic blocks cannot be precisely mapped due to optimizations, inlining, or missing/incomplete symbols, which may degrade the LLM context.

Future work could explore more robust source-level information recovery techniques to mitigate this limitation.

Precision of Indirect Call Resolution. To construct the call graph, Bulbasaur adopts Andersen-style points-to analysis [1] to resolve indirect calls. Although widely used in practice, this analysis may introduce a large number of false-positive call edges, potentially leading to inaccurate caller–callee relationships in the context provided to the LLM.

As future work, we will incorporate more efficient and precise pointer analyses (e.g., data-flow refinement [6, 27]) to improve call-graph accuracy.

7 Related Work

Solving hard branch constraints remains a fundamental challenge in coverage-guided fuzzing. Existing approaches fall into the following categories.

Symbolic and Concolic Execution. Symbolic execution [5, 20] and hybrid approaches like Driller [41] and QSYM [51] model program paths as symbolic formulas and use Satisfiability Modulo Theories (SMT) solvers to generate constraint-satisfying inputs. However, they struggle with complex constraints like CRC checks due to formula size explosion when symbolically executing library functions, solver timeouts on non-linear constraints, and the need for manual modeling of external libraries. ConcoLLMic [29] augments concolic execution with LLMs to assist in constraint solving, but still suffers from path condition accumulation as branch depth increases, limiting its effectiveness on deep or complex constraints.

In contrast, Bulbasaur leverages LLM-generated mutators to perform lightweight, targeted mutations on existing seeds, enabling efficient constraint solving with low overhead.

Taint-Guided Fuzzing. RedQueen [2], GREYONE [17], and PATA [25] employ taint tracking to identify input bytes affecting branch conditions and perform targeted mutations. While effective for simple equality constraints, these approaches still struggle with hard constraints where input-to-constraint relationships are highly non-linear (e.g., CRC32, MD5) or require coordinated modifications across multiple fields. Unlike these approaches, Bulbasaur leverages LLM-generated mutators to perform precise manipulations on relevant input bytes, enabling more reliable resolution of hard constraints.

LLM-enhanced Input Generation and Mutation. In recent years, a growing body of work [10] has explored the use of LLMs to assist input generation and mutation. Fuzz4All [46] targets language-processing systems such as compilers by synthesizing complete programs using LLMs. CovRL-Fuzz [15] applies coverage-guided reinforcement learning to LLM-based mutators for fuzzing JavaScript engines. MetaMut [34] focuses on automatically generating mutation operators by embedding domain knowledge into prompts to produce semantic-aware mutators. G2FUZZ [54] leverages LLMs to generate and mutate input generators for fuzzing programs with non-textual formats. Although these approaches improve the structural and semantic quality of generated inputs, they generally lack fine-grained awareness of branch constraints inside programs. As a result, they are unable to explicitly identify fuzzing bottlenecks and pass hard constraints.

8 Conclusion

This paper presents Bulbasaur, a branch-guided framework for online LLM-based mutator generation. By providing LLMs with rich static and dynamic program information, Bulbasaur enables the generation of high-quality mutators tailored to specific branch constraints, and adopts an efficient and adaptive strategy to apply these mutators for solving hard branch constraints during fuzzing. Evaluation results demonstrate that Bulbasaur outperforms state-of-the-art fuzzers and other mainstream approaches.

9 Ethical Considerations

This work studies the use of LLMs to assist fuzzing and vulnerability discovery. As such, it involves potential dual-use risks, as the techniques developed in this paper could be leveraged by both defenders and attackers.

Stakeholders. The primary stakeholders of this research include software developers and maintainers, end users of affected software systems, security researchers, and organizations responsible for vulnerability mitigation. Indirect stakeholders include vendors whose products may be affected by newly discovered vulnerabilities and users whose security may be impacted by delayed patch deployment.

Potential Harms. Our approach may accelerate the discovery of software vulnerabilities, which could potentially be exploited by malicious actors if disclosed irresponsibly. In addition, the automated nature of LLM-enhanced fuzzing may lower the barrier for vulnerability discovery, increasing the risk of large-scale exploitation. Furthermore, improper handling of vulnerability information may lead to premature disclosure and associated security risks. We acknowledge that adversarial misuse is an inherent dual-use risk of this work that cannot be fully prevented once the tool is publicly released.

Mitigation Measures. To mitigate these risks, we follow responsible disclosure practices for all discovered vulnerabilities. For each case reported in Section 5.7, we first confirmed reproducibility, then reported the issue through the corresponding project's GitHub issue tracker, and coordinated with maintainers to confirm the root cause and fix. The 15 vulnerabilities included in this paper represent only those for which maintainers responded and fixes were confirmed; additional bugs that received no response or remained unresolved were intentionally excluded to avoid unnecessary risk to affected projects and users. We refrained from releasing proof-of-concept exploits and avoided sharing sensitive details that could enable immediate exploitation. Our released artifacts do not include vulnerable inputs, exploit scripts, or detailed exploitation guidance.

Ethical Decision Process. We carefully evaluated the potential benefits and risks of this research before publication. While our techniques could be misused, we believe that the benefits to software security outweigh the potential harms. By enabling earlier detection and remediation of vulnerabilities, our work contributes to improving the overall security and reliability of widely used software systems. Throughout the research process, we followed established community norms and ethical guidelines for vulnerability research, including

coordinated disclosure and transparent communication with affected parties.

Future Considerations. We recognize that advances in LLM-assisted security analysis may further lower the cost of vulnerability discovery in the future. We therefore commit to continuously monitoring the societal impact of our techniques and adapting our disclosure and artifact-sharing practices accordingly.

Overall, we believe that our work was conducted and disseminated in an ethically responsible manner, balancing scientific advancement with the protection of affected stakeholders.

10 Open Science

Our artifact is publicly available on Zenodo: Artifact Link, including the source code of Bulbasaur, baseline fuzzers, and experimental scripts.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd for their constructive comments and efforts to help improve this paper. This work is supported by the National Natural Science Foundation of China under grant U24A20337 and by the Joint Research Center for System Security, Tsinghua University (Institute for Network Sciences and Cyberspace) – Science City (Guangzhou) Digital Technology Group Co., Ltd.

References

[1] Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Copenhagen, 1994.

[2] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with input-to-state correspondence. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium (NDSS)*, 2019.

[3] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. GRIMOIRE: Synthesizing structure while fuzzing. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1985–2002, 2019.

[4] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as a Markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1032–1043, 2016.

[5] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08)*, pages 209–224, 2008.

[6] Yuandao Cai, Yibo Jin, and Charles Zhang. Unleashing the power of type-based call graph construction by using regional pointer information. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 1383–1400, 2024.

[7] Olivier Chapelle and Lihong Li. An empirical evaluation of thompson sampling. *Advances in Neural Information Processing Systems*, 24, 2011.

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

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

[10] Yiyang Chen, Nuoqi Gui, Long Wang, Longfei Chen, Xuanqing Shi, Xi Cao, and Chao Zhang. TrigFuzz: Triggering Conditions Guided Directed Fuzzing . In *2026 IEEE Symposium on Security and Privacy (SP)*, pages 3357–3375, Los Alamitos, CA, USA, May 2026. IEEE Computer Society.

[11] Yiyang Chen, Chao Zhang, Long Wang, Wenyu Zhu, Changhua Luo, Nuoqi Gui, Zheyu Ma, Xingjian Zhang, and Bingkai Su. IDFuzz: Intelligent directed grey-box fuzzing. In *34th USENIX Security Symposium (USENIX Security 25)*, pages 6219–6238, 2025.

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

[13] Ruiqi Dong, Fanke Tong, He Huang, Xiaogang Zhu, Xi Xiao, Shaohua Wang, Sheng Wen, and Yang Xiang. One mutation fits all: Exploring universal library fuzzing based on exogenous mutation. *IEEE Transactions on Dependable and Secure Computing*, 2025.

[14] Michael Eddington. Peach fuzzer. https://peachtech.gitlab.io/peach-fuzzer-community/. Accessed: 2026-01-25.

[15] Jueon Eom, Seyeon Jeong, and Taekyoung Kwon. Fuzzing JavaScript Interpreters with Coverage-Guided Reinforcement Learning for LLM-Based Mutation. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2024.

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

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

[18] Honggfuzz: A security-oriented, feedback-driven evolutionary fuzzer. https://honggfuzz.dev/. Accessed: 2026-01-25.

[19] Patrick Jauernig, Domagoj Jakobovic, Stjepan Picek, Emmanuel Stapf, and Ahmad-Reza Sadeghi. DARWIN: Survival of the fittest fuzzing mutators. In *Proceedings of the 30th Annual Network and Distributed System Security Symposium (NDSS)*, 2023.

[20] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[21] Konstantinos Kitsios, Marcel Böhme, and Alberto Bacchelli. On interaction effects in greybox fuzzing. arXiv preprint arXiv:2510.19984, 2025.

[22] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2123–2138, 2018.

[23] Jun Li, Bin Zhao, and Chao Zhang. Fuzzing: a survey. *Cybersecurity*, 1(1):6, 2018.

[24] Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, Peng Cheng, et al. UNIFUZZ: A holistic and pragmatic metrics-driven platform for evaluating fuzzers. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2777–2794, 2021.

[25] Jie Liang, Mingzhe Wang, Chijin Zhou, Zhiyong Wu, Yu Jiang, Jianzhong Liu, Zhe Liu, and Jiaguang Sun. Pata: Fuzzing with path aware taint analysis. In *2022 IEEE Symposium on Security and Privacy (S&P)*, pages 1–17. IEEE, 2022.

[26] Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, et al. Deepseek-v3.2: Pushing the frontier of open large language models. arXiv preprint arXiv:2512.02556, 2025.

[27] Dinghao Liu, Shouling Ji, Kangjie Lu, and Qinming He. Improving indirect-call analysis in LLVM with type and data-flow co-analysis. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 5895–5912, 2024.

[28] LLVM Project. Libfuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html. Accessed: 2026-01-25.

[29] Zhengxiong Luo, Huan Zhao, Dylan Wolff, Cristian Cadar, and Abhik Roychoudhury. Agentic concolic execution. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pages 1–19, 2026.

[30] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. MOPT: Optimized mutation scheduling for fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1949–1966, 2019.

[31] Valentin JM Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 47(11):2312–2331, 2019.

[32] Ruijie Meng, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury. Large language model guided protocol fuzzing. In *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS)*, 2024.

[33] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. Fuzzbench: an open fuzzer benchmarking platform and service. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 1393–1403, 2021.

[34] Xianfei Ou, Cong Li, Yanyan Jiang, and Chang Xu. The mutators reloaded: Fuzzing compilers with large language model generated mutation operators. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 298–312, 2024.

[35] Van-Thuan Pham, Marcel Böhme, Andrew E Santosa, Alexandru Răzvan Căciulescu, and Abhik Roychoudhury. Smart greybox fuzzing. *IEEE Transactions on Software Engineering*, 47(9):1980–1997, 2019.

[36] PNG Development Group. libpng: The official portable network graphics reference library. https://github.com/pnggroup/libpng. Accessed: 2026-01-25.

[37] LLVM Project. Sanitizer coverage. https://clang.llvm.org/docs/SanitizerCoverage.html. Accessed: 2026-01-25.

[38] Kostya Serebryany. OSS-Fuzz—Google's continuous fuzzing service for open source software, 2017.

[39] Dongdong She, Abhishek Shah, and Suman Jana. Effective seed scheduling for fuzzing with graph centrality analysis. In *2022 IEEE Symposium on Security and Privacy (S&P)*, pages 2194–2211. IEEE, 2022.

[40] Dongdong She, Adam Storek, Yuchong Xie, Seoyoung Kweon, Prashast Srivastava, and Suman Jana. Fox: Coverage-guided fuzzing as online stochastic control. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 765–779, 2024.

[41] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS)*, 2016.

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

[43] William R Thompson. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25(3/4):285–294, 1933.

[44] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Superion: Grammar-aware greybox fuzzing. In *Proceedings of the 41st IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 724–735. IEEE, 2019.

[45] Mingyuan Wu, Ling Jiang, Jiahong Xiang, Yanwei Huang, Heming Cui, Lingming Zhang, and Yuqun Zhang. One fuzzing strategy to rule them all. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)*, 2022.

[46] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. Fuzz4all: Universal fuzzing with large language models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 1–13, 2024.

[47] Yuchong Xie, Wenhui Zhang, and Dongdong She. Ztaint-havoc: From havoc mode to zero-execution fuzzing-driven taint inference. *Proceedings of the ACM on Software Engineering*, 2(ISSTA):917–939, 2025.

[48] Yupeng Yang, Shenglong Yao, Jizhou Chen, and Wenke Lee. Hybrid language processor fuzzing via LLM-based constraint solving. In *34th USENIX Security Symposium (USENIX Security 25)*, pages 6299–6318, 2025.

[49] Dongsong Yu, Yiyi Wang, Chao Zhang, Yang Lan, Zhiyuan Jiang, Shuitao Gan, Zheyu Ma, and Wende Tan. xfuzz: A flexible framework for fine-grained, runtime-adaptive fuzzing strategy composition. *Proceedings of the ACM on Software Engineering*, 2(ISSTA):69–91, 2025.

[50] Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. EcoFuzz: Adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2307–2324, 2020.

[51] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 745–761, 2018.

[52] Michał Zalewski. American fuzzy lop - whitepaper. https://lcamtuf.coredump.cx/afl/technical_details.txt. Accessed: 2026-01-25.

[53] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. The fuzzing book, 2019.

[54] Kunpeng Zhang, Zongjie Li, Daoyuan Wu, Shuai Wang, and Xin Xia. Low-cost and comprehensive non-textual input fuzzing with llm-synthesized input generators. arXiv preprint arXiv:2501.19282, 2025.

[55] Kunpeng Zhang, Xiaogang Zhu, Xi Xiao, Minhui Xue, Chao Zhang, and Sheng Wen. Shapfuzz: Efficient fuzzing via shapley-guided byte selection. In *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS)*, 2024.

[56] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. Fuzzing: a survey for roadmap. *ACM Computing Surveys (CSUR)*, 54(11s):1–36, 2022.

A Prompt Templates

A.1 Project Description Prompt

Issued once before fuzzing begins. The LLM explores the target program via the program query agent and produces

a concise summary of its input formats, key constants, and parsing logic, which is prepended to all subsequent mutator generation prompts.

```
You are helping with fuzzing a target program.
Target: <program_name>
Command: <program_name> <exec_args>

[If documentation is provided:]
Project documentation:
<documentation content>

You have access to the following tools to explore
the project codebase:
  read_source_context / list_directory_files / grep
  query_caller / query_callee [if callgraph available]
  [... detailed usage instructions for each tool ...]

Explore the codebase as needed (up to 10 queries),
then provide a concise summary covering:
  1. What the program does and what it processes
  2. Supported input formats and key data structures
  3. Important constants, magic numbers, or checksums
  4. Key parsing and validation logic that affects
     how inputs are interpreted or rejected

This summary will be used to guide mutation of inputs
during fuzzing. Focus on information useful for
understanding how the program consumes its input.
Provide ONLY the summary, without meta-commentary.
```

A.2 Mutator Generation Prompt

Issued once per hard branch during fuzzing. Given the project description, branch/edge source context, and the operand-aware mutator template, the LLM generates a tailored mutation function via multi-round tool interaction.

```
[Project Description Summary]
---
You are assisting a coverage-guided fuzzing system.
The fuzzer has already discovered inputs that reach
Branch B, but cannot yet traverse Edge E from it.
Your task is to generate a targeted mutation function
that increases the probability of traversing Edge E
by modifying the input buffer.

BRANCH B (current position, already reachable)
- Location: <branch_file>:<branch_line>
- Function: <branch_function>
- Source context:
  <branch source code (+/-10 lines)>

EDGE E (target, not yet covered)
- Location: <edge_file>:<edge_line>
- Source context:
  <edge source code (+/-10 lines)>

TASK
Generate a mutation function that takes:
  buf -- the input buffer to mutate (Vec<u8>)
  op1, op2 -- runtime values of branch operands
[... operand-aware mutator template shown in Fig. 4 ...]
and returns 1 if the buffer was modified, 0 otherwise.

Since a seed already reaches Branch B, the function
should focus only on satisfying the condition needed
to traverse Edge E, without corrupting the prefix
path conditions already satisfied by the seed.

MUTATION GUIDELINES
- Analyze what condition at Branch B blocks Edge E.
- Determine which bytes in the input influence that
  condition (by reasoning over the source context
  and input format from the project description).
- When op1/op2 are informative, use them as anchors
  to locate or rewrite the relevant input region.
- Prefer structure-aware, semantics-guided mutations
  over arbitrary byte flips.

ADDITIONAL CONTEXT
You may request more source-level information via:
  read_source_context / list_directory_files / grep
  query_caller / query_callee [if callgraph available]
  [... detailed usage instructions for each tool ...]

INPUT-INDEPENDENT BRANCHES
If the branch condition cannot be influenced by
mutating the input, output:
  UNABLE_TO_BREAK_THROUGH
```

B LLM Cost Comparison of Bulbasaur, G2FUZZ, and Bulbasaur-G2FUZZ

We analyze the LLM cost of Bulbasaur, G2FUZZ, and Bulbasaur-G2FUZZ, as summarized in Table 9. Overall, Bulbasaur incurs higher API costs than G2FUZZ on 9 out of the 10 targets, except for `flvmeta`. This is primarily because G2FUZZ performs program-agnostic generator synthesis and mutation based on predefined input formats, resulting in relatively stable and low LLM usage across different programs. In contrast, Bulbasaur generates tailored mutators for hard branches, and its LLM cost therefore depends on the number of hard branches encountered during fuzzing. As a result, Bulbasaur exhibits higher overhead on targets with a large number of complex branches, such as `ffmpeg` and `pdftotext`.

For Bulbasaur-G2FUZZ, the LLM cost is higher than that of Bulbasaur, since it additionally invokes LLMs to generate and mutate input generators, leading to increased API usage.

Table 9: API cost incurred by Bulbasaur-G2FUZZ, Bulbasaur, and G2FUZZ on UniFuzz.

Target	Bulbasaur-G2FUZZ	Bulbasaur	G2FUZZ
exiv2	$6.96	$5.98	$0.91
flvmeta	$3.29	$0.54	$1.38
ffmpeg	$7.68	$6.29	$0.50
imginfo	$8.54	$6.49	$1.10
jhead	$2.41	$0.67	$0.58
gdk	$8.32	$5.79	$1.57
pdftotext	$7.16	$6.48	$0.80
tiffsplit	$5.93	$5.67	$0.55
mp3gain	$3.21	$2.50	$0.79
mp42aac	$7.05	$5.48	$1.18
Sum	$60.55	$45.89	$9.36

Table 10: Effectiveness of LLM-generated mutators for branch solving.

Program	LB	SB	SR
cflow	183	3	1.64%
exiv2	349	27	7.74%
flvmeta	19	0	0.00%
ffmpeg	361	36	9.97%
imginfo	355	34	9.58%
infotocap	109	5	4.59%
jhead	24	1	4.17%
jq	337	2	0.59%
gdk	234	11	4.70%
lame	386	8	2.07%
mujs	430	83	19.30%
nm	278	19	6.83%
objdump	267	36	13.48%
pdftotext	344	49	14.24%
sqlite3	445	103	23.15%
tcpdump	478	52	10.88%
tiffsplit	222	41	18.47%
mp3gain	95	5	5.26%
mp42aac	311	23	7.40%
Sum	5227	538	10.29%

LB: LLM Branches; **SB:** Solved Branches; **SR:** Solving Rate (SB / LB).

C Effectiveness of Branch Solving

To analyze the effectiveness of LLM-generated mutators for branch solving, we further examine, across 19 UniFuzz targets, the number of branches that trigger LLM-based mutator generation (LB), the number of branches successfully solved by LLM-generated mutators (SB), and the ratio of solved branches to LLM branches (SR), as summarized in Table 10.

Overall, Bulbasaur selects 5,227 branch constraints for mutator generation across all targets, among which 538 are successfully solved, yielding an overall solving rate of 10.29%. This result demonstrates that the generated mutators are effective in resolving a portion of hard branch constraints.

At the target level, we observe relatively high solving rates on `sqlite3` (23.15%) and `mujs` (19.30%). Both targets accept text-based inputs, where `sqlite3` processes SQL scripts and `mujs` interprets JavaScript code. This indicates that LLM-generated mutators are particularly effective for text-based programs, which aligns with the strong natural language and code generation capabilities of LLMs.

Meanwhile, Bulbasaur also achieves non-trivial solving rates on binary-based targets such as `nm` (6.83%) and `objdump` (13.48%), suggesting that our approach remains effective beyond purely textual inputs. These results demonstrate that Bulbasaur is broadly applicable to diverse types of programs, including both text-based and binary-based targets.

We also observe that Bulbasaur achieves relatively low solving rates on some targets, such as `jq` (0.59%) and `flvmeta` (0%). We further investigate the underlying reasons.

`jq` is a JSON query language interpreter. Under the default execution command in UniFuzz, it is executed with a fixed query, and fuzzing is performed only on the JSON input. As a result, most executions primarily exercise the JSON parsing logic and a limited subset of the query evaluation functionality, covering only a small fraction of `jq`'s overall capabilities. Consequently, most uncovered branches are not input-dependent, and only a small fraction of branches can be influenced by input mutations. As a result, the solving rate on `jq` remains low. This observation is consistent with the coverage results in Table 3, where different fuzzers achieve similar coverage on `jq`.

`flvmeta` is a finite-state-machine-based FLV parser. Our analysis shows that most LLM branches correspond to defensive state checks or error-handling paths, whose constraints cannot be satisfied under normal parsing logic. As a result, none of the 19 LLM branch constraints in `flvmeta` is successfully solved. Moreover, as shown in Table 3, most fuzzers reach a stable line coverage of 936 on `flvmeta`, indicating that the program has already reached a coverage plateau.

D Details of Discovered CVEs

Table 11: Real-world vulnerabilities discovered by Bulbasaur across OSS-Fuzz projects.

CVE ID	Project	Vulnerability Type
CVE-2025-2368	WABT	Heap-based Buffer Overflow
CVE-2025-2584		Heap-based Buffer Overflow
CVE-2025-3122		NULL Pointer Dereference
CVE-2025-2588	Augeas	NULL Pointer Dereference
CVE-2025-2912	HDF5	Heap-based Buffer Overflow
CVE-2025-2913		Use After Free
CVE-2025-2914		Heap-based Buffer Overflow
CVE-2025-2915		Heap-based Buffer Overflow
CVE-2025-2923		Heap-based Buffer Overflow
CVE-2025-2924		Heap-based Buffer Overflow
CVE-2025-2925		Double Free
CVE-2025-2926		NULL Pointer Dereference
CVE-2025-3010	Glslang	NULL Pointer Dereference
CVE-2025-3196	Assimp	Stack-based Buffer Overflow
CVE-2025-3548		Heap-based Buffer Overflow