香　港　大　學

THE UNIVERSITY OF HONG KONG

Fuzzing with Hopper++: Applying Large Language Models to Program Generation, Automatic Testing, and Security Evaluation

Laidlaw Scholar: Wang Hanyu

School of Computing and Data Science
The University of Hong Kong

Research Supervisor: Prof. Chen Ho

Chair Professor of Musketeers Foundation Institute of Data Science
and Department of Computer Science

August 2025

Abstract

The focus of this research was to leverage the power of Generative AI and apply Large Language Model (LLM) to fuzz testing, a software security testing technique that inputs random data into a program to trigger potential crashes.

Based on the previous interpretative fuzzing project Hopper (Chen et al., 2023), this work introduces Hopper++, an enhanced version that incorporates an LLM module to automate four key processes: (1) fuzzing input generation with Domain-Specific-Language (DSL), (2) automatic fuzzing with LLM targeted on specific API, (3) API constraint synthesis with DSL, and (4) intelligent constraint merging and updating. This augmentation significantly extends Hopper's capabilities in library fuzzing while maintaining its core interpretative fuzzing architecture.

1 Introduction

1.1 Research Background

1.1.1 Fuzz Testing

In software development and security engineering, fuzz testing is an automated software testing technique that involves feeding random, unexpected, or malformed data inputs into a program to trigger crashes, memory leaks, or security vulnerabilities [1].

Ever since this concept was first introduced by Barton Miller and his team in the 1990s, fuzzing has kept evolving throughout the years with modern variants such as coverage-guided fuzzing [2] and semantic-aware fuzzing [3]. Contemporary fuzzing frameworks typically employ either mutation-based strategies, which alter existing valid inputs, or generation-based approaches that construct test cases from formal specifications.

Some of the well-developed and widely-used fuzzing tools include American Fuzzy Lop (AFL) [4], LibFuzzer [5], and Honggfuzz [6].

1.1.2 Library Fuzzing

Fuzzing libraries is of great significance in the field of software security, as they are often vulnerable points in the system. Among other techniques, greybox fuzzing has made outstanding contributions in the field and possesses exciting potential. It adopts lightweight instrumentation and coverage-guided energy assignment methods to guide input mutation toward unexplored code paths without requiring semantic knowledge about the target program, improving the possibility of detecting deep and hidden vulnerabilities.

However, applying these benefits to library fuzzing presents unique challenges. Unlike binary fuzzing, where inputs are processed by predefined entry points, library fuzzing requires fuzz drivers to transform byte streams and correctly invoke APIs. Unfortunately, on one hand, writing manually-crafted fuzz drivers can be

extremely demanding in terms of background knowledge of the program itself. Even if fuzz drivers are already included in the library, they can often only cover a relatively small part of the APIs, making it incompetent to discover vulnerabilities for the whole library. On the other hand, automatically-generated fuzz drivers rely heavily on existing consumer code, which can be absent or misleading sometimes, leading to false invocation of the target library's API.

1.1.3 Hopper

Recognizing these challenges in library fuzzing, an interpretative fuzzing tool, Hopper, was proposed to help with addressing the problem. Hopper reformulates library fuzzing as interpreter fuzzing by incorporating a lightweight DSL interpreter to dynamically translate generated inputs into API calls. Additionally, it also possesses API constraint learning capabilities through grammar-aware mutation and dynamic feedback, correctly inferring the vast majority of intra- and inter-API constraints without external guidance.

1.2 Research Objectives

Despite its advantages in library fuzzing, Hopper still exhibits several limitations in its current design. To begin with, Hopper generates input test cases by first selecting a target function and then constructing statements and arguments to form a program around it. This function-centric, code-based approach may overlook the invocation context and the logical relationships among different APIs. Another potential shortcoming is that rule-based generation followed by minor mutations limits flexibility and creativity in forming API call sequences, which restricts the depth of fuzzing and reduces the likelihood of triggering crashes caused by unusual API usage patterns.

In light of that, we introduced an LLM-based DSL generator in the new version of Hopper, leveraging generative AI to produce innovative test cases that complement human-crafted inputs. Additionally, during runtime, when Hopper

discovers an API that it has failed to learn constraints from and has a high probability of crashing spuriously, it will no longer generate inputs for that API. However, these risky APIs are often crucial for uncovering program vulnerabilities. Therefore, Hopper's previous strategy of avoiding fuzzing such APIs inadvertently misses valuable opportunities to detect real crashes. To address this issue, our work will introduce a target-oriented phase that leverages LLM capabilities to specifically focus on these problematic APIs and fuzz them thoroughly. Lastly, false positive cases in API constraint learning often occur as approximations of the ground truth constraints. Constraints inferred by trial-and-error during runtime frequently lack accuracy and precision, resulting in an incorrect understanding of the library's API invocation rules. To overcome this limitation, we propose adding a new step of LLM-generated constraint DSL to serve as a supplement to runtime inferred constraints. Both sets of constraints will be merged using carefully designed rules to ensure that only the highest-quality constraints are retained in Hopper's log file.

2 Methodology

2.1 Overview

In the new version of Hopper++, we integrated an LLM module into the original system through the following process. After compilation and instrumentation, Hopper enters the fuzzing stage, which consists of a pilot phase followed by an evolution phase. Concurrently, the LLM module is activated. It begins with a generation phase, where LLM will craft DSL test cases after learning from the manually constructed representative examples, ensuring grammatical correctness. After multiple cycles of generation and fuzzing, the system typically reaches a performance plateau with a lack of new crashes and minimal updates to constraints over an extended period. At this point, the LLM module transitions to a target-oriented phase. In this phase, previous crash and hang cases will be thoroughly analyzed, and attention is focused on problematic APIs, particularly those that remain underexplored despite extensive mutation of related test case inputs. Throughout the entire fuzzing process, all API constraints inferred are continuously updated in the log file and reconciled through a merging strategy that incorporates both rule-based and LLM-driven decisions.

2.2 Generation Phase

The generation phase aims at creating AI-generated DSL test cases with innovative API call sequences to enhance fuzzing effectiveness beyond manual testing. In this phase, the input prompt for LLM is composed of four key elements:

1. **Grammar rules** for the test case DSL provide high-level instructions to ensure syntactically valid inputs. Each DSL program is built from statements, which serve as fundamental units, each assigned a unique index that can be referenced by subsequent statements. Five common types of statements are identified in DSL, corresponding to typical fuzzing actions: load, call, update, assert and file. Each statement is then assigned expansions of various

values, yielding a diverse set of test case inputs. (Chen et al., 2023)

2. **Manually selected representative test case examples** are included to guarantee that the LLM-generated DSL conforms to correct grammar. These examples are general cases that are not tied to specific libraries or APIs, encouraging LLM to exercise transfer learning so that it can apply the grammar learnt across a broad spectrum of libraries.

3. **Library documentations** and header files to provide LLM with essential background knowledge of available APIs to invoke and required argument types for each function.

4. **Strict generation guidelines** are imposed on the LLM to prevent common grammatical errors and invalid API call patterns, ensuring the quality and correctness of the generated test inputs.

2.3 Target-Oriented Phase

The target-oriented phase differs from the generation phase primarily because it concentrates on specific APIs rather than the entire library. Target APIs are selected from two sources:

1. **Crashes and hangs** captured during execution. This phase serves the initial purpose of reducing false positive crash cases by verifying whether they violate known constraints. If crashes or hangs result from incorrect API invocations that do not adhere to existing constraints, these are classified as false positives and not considered true vulnerabilities. Once a crash is confirmed as genuine, the API calls within the corresponding test case input become individual fuzzing targets. The goal is to isolate the root cause of the crash and identify it as a valid vulnerability within the library.

2. **APIs that remain insufficiently fuzzed** despite extensive mutations of related test inputs. Instead of performing line-by-line code analysis, we tend

to adopt a lightweight black-box approach guided by heuristic principles. The system utilizes LLM to statistically infer potential targets by detecting whether a fuzzing harness aimed at a particular API repeatedly results in failures or problematic outcomes. Specifically, by monitoring test cases that undergo numerous mutation cycles but fail to increase API coverage, or equivalently, are never selected as parents for producing the next generation of DSL inputs, we can conclude that the associated API has not been thoroughly explored and thus deserves focused attention.

Relavant test cases from execution records and constraints from log files related to the target API are input into the LLM to guide fuzzing toward deeper coverage.

2.4 API Constraints

API constraints are essential outcomes of Hopper's fuzzing process and play a critical role in guiding subsequent fuzzing iterations. These constraints originate from three main sources: (1) user-customized inputs, (2) runtime dynamic inference by Hopper, and (3) constraints generated by LLM. Given that the same API constraint may be inferred multiple times across different sources or runtime rounds but with slight variations in conditions, robust merging rules are essential to determine the most accurate version for updating Hopper's log file.

The original version of Hopper included basic merging rules for straightforward constraint types, such as Range, NonZero, SetVal, File, Arraylength and LengthFactor.Due to their well-defined and certain conditions, these constraints are suitable for adopting rule-based methods with hard-coded logic. However, more complex constraints, for instance, CastFrom, RetFrom, UseUnionMember, InitWith, Context and OpaqueType, have not been handled previously due to their complicated nature.

In this research, we leveraged the power of LLM to implement a comprehensive constraint merging strategy to maintain the accuracy and effectiveness of constraint recording during the fuzzing process. For each constraint type, spe-

cialized merging rules are embedded in the LLM prompt, with outputs structured in JSON format specifying actions, whether to keep the existing constraint, use the new constraint, or merge two constraints into another one, alongside explicit reasoning for each decision.

Take the merge of the CastFrom constraint as an example. We prioritize specificity by retaining more concrete cast types over general ones such as void*, while ensuring type compatibility and respecting inheritance relationships within the C/C++ type system. Additionally, user-defined constraints take precedence over system-inferred ones. When conflicts arise, multiple constraints may be retained if necessary to accommodate practical scenarios. Our approach is context-aware, considering runtime conditions such as pointer states and modifications, to avoid unsafe casts that could lead to crashes.

3 Outcome

Based on comparative analysis of the fuzzing outcome statistics between the two versions, Hopper++ demonstrates clear performance improvements over the original version in three key aspects: (1) direct fuzzing performance logged during runtime, (2) the quality of the crashes detected, and (3) the effective amount of API constraints inferred.

This analysis draws on experimental fuzzing statistics collected from using both Hopper and Hopper++ to fuzz the same library, cJSON, under identical time and machine environment conditions.

3.1 Fuzzing Performance

```
Hopper:
INFO [hopper-core/src/fuzzer.rs:560] #queue: 3338, #crashes: 4, #hangs: 0, #edge: 3504,
    density: 5.34%, #round: 20.45k, #exec: 4.88m (57\%), #speed: 2457.71 (0.23ms)

Hopper++:
INFO [hopper++ -core/src/fuzzer.rs:560] #queue: 3867, #crashes: 4, #hangs: 0, #edge: 3553,
    density: 5.42%, #round: 19.45k, #exec: 4.66m (67%), #speed: 2344.37 (0.28ms)
```

1. **Code Coverage**: Hopper++ achieves a higher code coverage with 3,553 edges discovered, surpassing Hopper's 3,504 edges by 49, indicating a more comprehensive exploration of the target program's codebase.

2. **Queue Size**: Hopper++ contains a larger queue size of 3,867 effective test cases, 529 more than Hopper's 3,338, reflecting enhanced input diversity and mutation effectiveness.

3. **Executor Usage Rate**: Hopper++ shows superior executor utilization with a 67% usage rate, 10 percentage points higher than Hopper's 57%, demonstrating more efficient resource usage during fuzzing.

4. **Round Efficiency**: Hopper++ attains these advancements with fewer rounds, 19.45k compared to Hopper's 20.45k rounds, demonstrating higher round efficiency by achieving better results with less iteration.

Collectively, these improvements indicate that Hopper++ provides deeper and faster code exploration with enhanced resource management, thus representing a substantial advancement in fuzzing effectiveness and efficiency over its original version.

3.2 Crash Case Quality

```
Hopper:
INFO [hopper_sanitizer] Remove the duplicated crashes by rip..
INFO [hopper_sanitizer] sanitize 4 crashes to : 4 by rip
INFO [hopper_sanitizer] Remove the duplicated crashes from crash groups ...
WARN [hopper_sanitizer] duplicated crashes `"Hopper/output/crashes/id_000002"` found in groups cJSON_DetachItemViaPointer
WARN [hopper_sanitizer] duplicated crashes `"Hopper/output/crashes/id_000003"` found in groups cJSON_DetachItemViaPointer
INFO [hopper_sanitizer] sanitize 4 crashes to : 2 by grouped sanitize
INFO [hopper_sanitizer] Remove the duplicated crashes by pc of clang sanitizer..
INFO [hopper_sanitizer] Sanitize the crashes caused by variable length arugments ...
INFO [hopper_sanitizer] sanitize 2 crashes to : 2 by filtering out the variable argument crashes.
INFO [hopper_sanitizer] Infer the crashes again ...
INFO [hopper_sanitizer] Save crashes to minimized crashes directory ...
INFO [hopper_sanitizer] classify 2 crashes into: 2 infered and 0 uninfered

Hopper++:
INFO [hopper_sanitizer] Remove the duplicated crashes by rip..
WARN [hopper_sanitizer] duplicated crash filtered out by rip: 0x789093e03cb0
INFO [hopper_sanitizer] sanitize 4 crashes to : 3 by rip
INFO [hopper_sanitizer] Remove the duplicated crashes from crash groups ...
INFO [hopper_sanitizer] sanitize 3 crashes to : 3 by grouped sanitize
INFO [hopper_sanitizer] Remove the duplicated crashes by pc of clang sanitizer..
INFO [hopper_sanitizer] Sanitize the crashes caused by variable length arugments ...
INFO [hopper_sanitizer] sanitize 3 crashes to : 3 by filtering out the variable argument crashes.
INFO [hopper_sanitizer] Infer the crashes again ...
INFO [hopper_sanitizer] Save crashes to minimized crashes directory ...
INFO [hopper_sanitizer] classify 3 crashes into: 3 infered and 0 uninfered
```

Although both Hopper and Hopper++ triggered the same number of crashes initially, Hopper++ demonstrates significant improvements in crash quality through its introduction of LLM-generated test case inputs.

Hopper++ leverages LLM to generate highly diverse and semantically rich inputs, leading to exploration of deeper and more varied code paths, which in turn produces higher quality crashes. This is reflected in the crash sanitizer pipeline where Hopper++ reduces the initial 4 crashes to 3 high-quality crashes after RIP deduplication, maintaining all after group deduplication, and infers violated constraints for all of them.

In contrast, Hopper started with 4 crashes but after group deduplication was reduced to 2, though both managed to infer violated API constraints for their respective crashes. The deeper semantic understanding embedded in the LLM-

generated inputs allows Hopper++ to uncover more informative and actionable crash instances, improving the value of fuzzing outputs for API usage constraint analysis and bug detection. Thus, Hopper++'s advantage in crash quality is primarily driven by advancements in test input generation through large language models rather than changes in post-crash processing.

3.3 Inferred Constraints

The quantifiable improvement in Hopper++ over Hopper is also evidenced by the increased number of inferred API usage constraints. Hopper++ infers 78 constraints, surpassing Hopper by two additional constraints.

The two new constraints correspond to specific functions in cJSON library: `cJSON_SetNumberHelper` and `cJSON_AddNumberToObject`. Both functions have detailed argument constraint profiles in Hopper++, reflecting a deeper understanding of their API usage patterns. For example, in constraints inferred by Hopper++, `cJSON_SetNumberHelper` is characterized by non-null argument constraints and precise argument grouping, while `cJSON_AddNumberToObject` shows complex roles including both consumer and producer relationships in its return type. These added constraints indicate Hopper++'s enhanced capability to detect subtle and complex API usage conditions, thus providing richer semantic insights into potential API misuse or vulnerabilities. This further demonstrates Hopper++'s superior fuzzing outcome quality, as it extracts more comprehensive and actionable inference from these crash cases compared to the original Hopper.

```
func cJSON_SetNumberHelper = { can_succeed: T, insert_fail: F, internal: F, arg_constraints:
    vec(2)[{ list: vec(1)[{ key: [], constraint: NonNull$,  }, ],  }, { list: vec(0)[],  }, ],
    arg_group: vec(2)[0, 1, ], contexts: vec(0)[], role: { init_arg: F, free_arg: F,  },
    ret: { is_opaque: F, is_static: F, is_unwriteable: F, is_partial_opaque: F, both_cosumer_and_producer: F,  },  }

func cJSON_AddNumberToObject = { can_succeed: T, insert_fail: F, internal: F, arg_constraints:
    vec(3)[{ list: vec(0)[],  }, { list: vec(0)[],  }, { list: vec(0)[],  }, ], a
    rg_group: vec(3)[0, 1, 2, ], contexts: vec(0)[], role: { init_arg: F, free_arg: F,  },
    ret: { is_opaque: F, is_static: F, is_unwriteable: F, is_partial_opaque: F, both_cosumer_and_producer: T,  },  }
```

4 Conclusion

In conclusion, this research presents Hopper++, an enhanced library fuzzing framework that integrates large language models to improve key aspects of the fuzzing process.

By including LLM-driven DSL generation, target-oriented fuzzing phase, and comprehensive constraint merging strategy, Hopper++ effectively overcomes critical limitations of its previous version, enabling deeper exploration of APIs and more accurate constraint inference. Our approach reduces false positives and uncovers vulnerabilities that manual or rule-based methods might miss, demonstrating the power of combining AI and software security testing.

While challenges remain, such as expanding the framework's scope and scaling it up to a wider range of complex libraries, Hopper++ sets a promising direction for intelligent and adaptive fuzzing procedures for software security.

Acknowledgment

I would like to express my sincere gratitude to Lord Laidlaw and the Laidlaw Foundation for funding this invaluable research opportunity. It has been a great honor to be a Laidlaw Scholar at The University of Hong Kong. I am deeply thankful to my supervisor, Prof. Chen Ho, as well as Dr. Chen Peng and Senior Fellow Lyu Yunlong from the HKU Musketeers Foundation Institute of Data Science, for their dedicated guidance and support.

I truly cherish this opportunity, viewing it not merely as a summer research project, but as the beginning of a long-term research commitment. This experience marks my first significant step into the field of scientific research.

References

[1] V. J. M. Manès, C. Demetrescu, D. Malerba, and C. Mu, "The art, science, and engineering of fuzzing: A survey," *IEEE Security & Privacy*, vol. 17, no. 6, pp. 14–23, 2019.

[2] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, (New York, NY, USA), p. 2329–2344, Association for Computing Machinery, 2017.

[3] P. Godefroid, H. Peleg, and R. Singh, "Learnfuzz: machine learning for input fuzzing," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ASE '17, p. 50–59, IEEE Press, 2017.

[4] M. Zalewski, "American fuzzy lop." `http://lcamtuf.coredump.cx/afl/`, 2014. Accessed: 2023-08-23.

[5] R. Hodován, G. Ács, and Á. Tóth, "Libfuzzer: A coverage-guided fuzzing engine," in *Proceedings of the 2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, (New York, NY, USA), pp. 58–59, ACM, 2018.

[6] I. Nikolić and B. Seefeld, "Honggfuzz: Security oriented, feedback-driven, evolutionary fuzzing." `https://github.com/google/honggfuzz`, 2016. GitHub repository.