Testing like Mad Libs: Fuzzing SMT Solvers with Historical Unusual Inputs Empowered by LLMs

MAOLIN SUN, State Key Laboratory for Novel Software Technology, Nanjing University, China
YIBIAO YANG*, State Key Laboratory for Novel Software Technology, Nanjing University, China
HAOXIANG JIA, School of Computer Science, Peking University, China
JIANGCHANG WU, State Key Laboratory for Novel Software Technology, Nanjing University, China
QINGYANG LI, State Key Laboratory for Novel Software Technology, Nanjing University, China
ZIFAN XIE, Chongqing University, China
MING WEN, Huazhong University of Science and Technology, China
YUMING ZHOU, State Key Laboratory for Novel Software Technology, Nanjing University, China

Ensuring the correctness of Satisfiability Modulo Theory (SMT) solvers is of paramount importance, as it serves as the cornerstone of a broad spectrum of critical software engineering practices. Consequently, various methodologies have been proposed to test SMT solvers, including recent advances that utilize Large Language Models (LLMs) to generate input formulas for SMT solver fuzzing. However, directly employing LLMs to craft SMT formulas will result in an abundance of invalid inputs. Furthermore, our recent study demonstrated the effectiveness of using historical bug-triggering inputs to guide the generation of test formulas, as these inputs help exercise solvers' deep states. Building on these insights, we present MadFuzz, an enhanced approach that leverages the capabilities of LLMs to ingeniously synthesize test formulas by completing the skeletons of historical bug-triggering inputs. Moreover, MadFuzz incorporates a feedback mechanism based on solvers' behavior, which consists of LLM-guided prompt optimization and sampling-based seed input selection. To evaluate the effectiveness of MadFuzz, we conducted a comprehensive assessment using three cutting-edge SMT solvers: Z3, cvc5, and Bitwuzla. Our results demonstrate that MadFuzz has identified 20 confirmed real bugs in the solvers, 19 of which have already been addressed by developers. Additionally, our experiments demonstrate that MadFuzz surpasses existing SMT solver fuzzers in code coverage and bug detection capability.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: SMT solver, Fuzzing, Large Language Model, Skeleton

*Corresponding author.

Authors' Contact Information: Maolin Sun, State Key Laboratory for Novel Software Technology, School of Computer Science, Nanjing University, Nanjing, China, merlin@smail.nju.edu.cn; Yibiao Yang*, State Key Laboratory for Novel Software Technology, School of Computer Science, Nanjing University, Nanjing, China, yangyibiao@nju.edu.cn; Haoxiang Jia, Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education; School of Computer Science, Peking University, Beijing, China, haoxiangjia@stu.pku.edu.cn; Jiangchang Wu, State Key Laboratory for Novel Software Technology, School of Computer Science, Nanjing University, Nanjing, China, jiangchangwu@smail.nju.edu.cn; Qingyang Li, State Key Laboratory for Novel Software Technology, School of Computer Science, Nanjing University, Nanjing, China, liqingyang@smail.nju.edu.cn; Zifan Xie, Chongqing University, Chongqing, China, xzff@cqu.edu.cn; Ming Wen, Huazhong University of Science and Technology, Wuhan, China, mwenaa@hust.edu.cn; Yuming Zhou, State Key Laboratory for Novel Software Technology, School of Computer Science, Nanjing University, Nanjing, China, zhouyuming@nju.edu.cn.

ACM 1557-7392/2026/2-ART
https://doi.org/10.1145/3796526

1 Introduction

Satisfiability Modulo Theory (SMT) solvers are essential tools for automatically determining the satisfiability of logical formulas with respect to background theories, such as arithmetic, bit-vectors, and strings [3]. SMT solvers have a wide range of applications in practice, including software engineering [19, 23], computer security [40], and artificial intelligence [45]. For instance, the renowned symbolic execution engine KLEE [6] relies on SMT solvers to check the feasibility of program paths. However, bugs hidden in SMT solvers can inevitably mislead client applications, resulting in serious consequences and even security vulnerabilities [18]. Thus, ensuring the reliability of SMT solvers is of critical importance.

To this end, various approaches have been proposed for testing SMT solvers, which can be broadly classified into three categories: (1) *generation-based approaches* [5, 30], (2) *mutation-based approaches* [21, 27, 34, 39, 46, 47], and (3) *LLM-based approaches* [38, 48]. Generation-based approaches typically generate formulas from scratch based on predefined generation rules or grammar. Mutation-based approaches, on the other hand, involve mutating existing formulas to generate new ones. One of the most advanced approaches in this category is HistFuzz [39], which utilizes historical bug-triggering formulas to generate new formulas. HistFuzz first extracts the sequences of logical connectives and quantifiers in bug-triggering formulas as skeletons and then instantiates them with atomic formulas (i.e., the sub-formulas that cannot be further decomposed) to generate new formulas. Furthermore, researchers have explored LLM-based approaches inspired by the advancement of large language models (LLMs). These approaches aim to generate SMT formulas directly using LLMs [38, 48]. Fuzz4All [48] is a universal framework applicable to various software systems, including SMT solvers. It can generate test inputs for SMT solvers by prompting the generation LLM with the prompt generated by another LLM.

Despite the promising results demonstrated by the aforementioned approaches, they still exhibit several limitations. Specifically, generation-based methods encounter challenges in generating a diverse range of test formulas due to the inflexibility of established generative rules. Mutation-based techniques are constrained by their mutation spaces [34, 38], significantly influenced by the selected mutation strategies. For instance, our prior work, HistFuzz [39], presented in the conference version, relies solely on atomic formulas derived from bug-triggering formulas for test formula generation, potentially limiting its ability to generate diverse formulas compared to the bug-triggering formulas. Furthermore, current LLM-based approaches [38, 48] only straightforwardly employ LLMs for test formula generation. These methods direct LLMs to generate complete formulas from scratch, which often results in a substantial proportion of invalid inputs—nearly 50% [38, 48]—and even meaningless reports.[1] Furthermore, they overlook the critical role of formula skeletons in uncovering bugs [39]. Consequently, there is still room for improvement in existing approaches to fully harness the potential of LLMs and enhance their effectiveness in bug detection.

Our Approach. In this study, we propose MadFuzz, an enhanced approach that leverages LLMs to synthesize test inputs for fuzzing SMT solvers, drawing inspiration from Mad Libs.[2] Specifically, LLMs act as skeleton enumerators, generating test formulas by completing the ***skeletons*** of historical bug-triggering formulas. The historical bug-triggering formulas are sourced from the issue trackers of open-source SMT solvers such as Z3 [11] and cvc5 [1], and their skeletons are demonstrated to be effective in revealing new bugs [39]. To investigate the utility of these unusual formulas, we employ in-context learning [24], including zero-shot and few-shot learning, to prompt LLMs to generate test formulas with skeletons analogous to the bug-triggering formulas. Furthermore, we integrate a ***feedback mechanism*** to dynamically refine the generation process of MadFuzz, leveraging insights from solvers' behavior. The feedback mechanism includes two facets: (1) LLMs as optimizers to adjust prompts for the generator, and (2) *Markov Chain Monte Carlo* (MCMC) sampling [9] to modulate the selection

[1] https://github.com/Z3Prover/z3/issues/6818

[2] Mad Libs is a classic word game in which phrases are masked in a document, and someone unfamiliar with the document fills in the blanks to create a humorous story.

probability of bug-triggering formula skeletons. Concretely, on the one hand, if a bug-triggering formula's skeleton with the current prompt tends to generate test formulas with syntactic errors, MadFuzz utilizes another LLM to optimize the prompt immediately during the current iteration. On the other hand, we employ MCMC sampling to adaptively control the probability of selecting bug-triggering formula skeletons. Specifically, the skeletons of bug-triggering formulas tending to produce valid test formulas or uncover bugs are given a higher probability of selection for subsequent fuzzing loops, whereas those prone to generating invalid formulas are demoted in selection likelihood. Ultimately, MadFuzz applies each generated formula to the SMT solvers under test and identifies bugs via differential testing, which compares the results of different solvers on the same input.

We have implemented MadFuzz as a practical fuzzing tool for SMT solvers based on CodeLlama [37] and Llama-2 [42]. To evaluate the effectiveness of MadFuzz, we applied it to the leading SMT solvers, including Z3, cvc5, and Bitwuzla [29]. In addition to the bugs discovered by HistFuzz, MadFuzz has successfully identified 22 new bugs in the solvers, of which 20 were confirmed by the developers and 19 were fixed. Besides, our experiments also exhibit that MadFuzz outperforms advanced SMT solver fuzzers in terms of achieved code coverage and effectiveness.

Contributions. We make the following major contributions:

- ***Novelty:*** We introduce MadFuzz, an improved version of HistFuzz [39], that draws inspiration from the Mad Libs game to generate diverse SMT test inputs. MadFuzz leverages LLMs to synthesize test formulas based on historical bug-triggering formulas, combining the creativity of LLMs with the effectiveness of the underlying structure of historical bug-triggering formulas to enhance bug exposure.
- ***Originality:*** Our approach introduces versatile strategies to generate test formulas that share similar skeletons with bug-triggering formulas, offering a fresh perspective on software system testing. We posit that LLMs can inject new vitality into all skeleton-based testing techniques, and our work represents a step towards this direction.
- ***Technique:*** We implement our idea as a practical tool MadFuzz, leveraging the instruction-following capability of LLMs. To generate novel test formulas that can effectively trigger bugs, MadFuzz incorporates a feedback mechanism for dynamically optimizing the generation process.
- ***Practicality:*** Our tool MadFuzz has led to 22 bug reports for the leading SMT solvers Z3, cvc5, and Bitwuzla, of which 20 are confirmed and 19 are fixed by developers. Experimental results also demonstrate that MadFuzz outperforms the state-of-the-art SMT solver fuzzers, including our prior work HistFuzz, in terms of achieved code coverage.

This paper presents a substantial extension of our previous work, HistFuzz [17]. The extensions are threefold, encompassing significant methodological advances and a more comprehensive empirical assessment. First, we expand the original idea of reusing elements from historical bug-triggering formulas by incorporating LLMs to complete formula skeletons and generate new test inputs. This enhancement enables the creation of more diverse and expressive formulas than those produced in our earlier work. Second, we introduce a feedback-driven generation mechanism that guides the LLM during test creation. By using solver responses to refine subsequent iterations, this feedback mechanism improves the relevance and effectiveness of the generated formulas. Third, we significantly strengthen the evaluation. Specifically, we identify new bugs in several state-of-the-art systems, including Z3, cvc5, and Bitwuzla, and conduct a more comprehensive comparison between MadFuzz and recent testing approaches.

Paper Organization. The remainder of this paper is organized as follows. Section 2 provides an overview of the necessary background and describes related work. Section 3 formalizes our approach and describes the implementation of MadFuzz. Next, we elaborate on our extensive evaluation in detail in Section 4, and Section 5 dedicated to in-depth discussions on the results. The conclusion is presented in Section 6.

2 Background and Related Work

In this section, we first provide an overview of the SMT-LIB language and the standard for describing SMT formulas. Then, we introduce the large pre-trained language models.

2.1 SMT-LIB Language

The SMT-LIB standard language stands out as one of the most widely embraced input languages for SMT solvers [2]. Developed and championed by the SMT-LIB initiative, this standard represents a global effort to streamline research and foster development in the field of SMT. The SMT-LIB standard language specifies the syntax and semantics of the inputs for SMT solvers, as well as the specifications of background theories and logics. The most basic commands in the SMT-LIB language are `declare-fun`, `assert`, and `check-sat`. The `declare-fun` command is used to declare new symbols. In the SMT-LIB language, variables and functions can be declared in a uniform way. The difference is that variables are treated as functions without arguments. For example, the statement "(`declare-fun` x () Int)" declares the variable x as an integer type. The `assert` command adds constraints to the variables or functions that the formula should satisfy, such as the "(`assert` (and ($<$ x 3) ($>$ y 1)))" statement asserts that variable x is less than 3 and variable y is larger than 1. The `check-sat` statement queries the solver to decide on the satisfiability of a formula. Typically, an SMT instance includes multiple `assert` statements. They can be seen as the conjunctions of multiple constraints. Therefore, an SMT solver will return `sat` only if the constraints can be satisfied simultaneously. In contrast, the instance is deemed `unsat` if no assignment can satisfy all assertions.

2.2 SMT Solver Fuzzing

SMT solvers have become significant tools in software engineering practice, as evidenced by their widespread adoption and critical roles across various applications [13, 28, 41, 43]. In response to the vital need for reliable SMT solvers, various fuzzing techniques have been developed. Among the pioneering efforts is FuzzSMT by Brummayer and Biere [5], a grammar-based blackbox fuzzing tool. Subsequent advancements include Murxla [30], a model-based fuzzer designed to generate API call sequences for SMT solver validation. Despite their innovations, these approaches rely on expert-crafted grammars or models, potentially limiting the diversity and effectiveness of the test formulas generated. In addition to these generation-based techniques, mutation-based techniques have also been explored, revealing promise in bug detection for SMT solvers. Winterer et al. [47] obtain satisfiability-preserving formulas by synthesizing two formulas using the method of semantic fusion. STORM [27] generates satisfiable formulas with Boolean structures different from the original seed through construction. OpFuzz [46] tests SMT solvers by simply mutating operators and achieves satisfactory results. Typefuzz [34] improves OpFuzz by combining mutation and generation to produce more diverse mutants. Despite achieving promising results, these methods overlook the unique potential of historical bug-triggering formulas for uncovering new bugs. To address this gap, our earlier work, HistFuzz [39], introduced in the conference version, utilizes the skeletons of historical bug-triggering formulas to generate new test cases with the atomic formulas from these formulas. However, its reliance on bug-triggering formulas shares the mutation-based approaches' limitations in diversity, hampering exploration of the mutation space. In this journal extension, we propose to leverage LLMs to expand the mutation space and sufficiently exploit the potential of historical bug-triggering formulas to unearth new bugs.

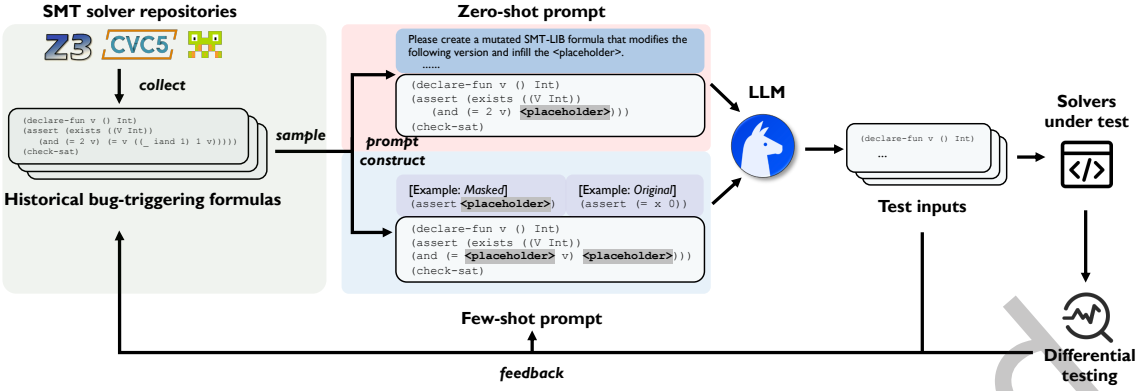

Fig. 1. Overview of MadFuzz.

2.3 Large Language Model

Transformer-based LLMs have revolutionized natural language processing (NLP), showcasing exceptional performance in tasks like text generation and question answering [42]. Trained on extensive corpora, these models have utility extending to software engineering practice, aiding in code generation, completion, and summarization [44], as well as software test case generation [14, 15, 38, 44, 48, 53]. A comprehensive review by Zhu et al. [58] also summarizes the applications of LLMs in software security domains such as fuzzing and program repair. In the context of fuzzing, Ye et al. [53] propose a fuzzing technique that fine-tunes GPT-2 to generate test inputs for JavaScript engines. More recently, Deng et al. [14] utilized LLMs to fuzz deep learning libraries. Their approach involves employing a generative model, Codex, to generate programs that invoke deep learning libraries, and further mutate the generated programs. In addition to leveraging LLMs in a zero-shot manner, Deng et al. [15] also propose fine-tuning LLMs specifically for deep learning library fuzzing. Beyond these domains, LLM-based fuzzing has been applied to other complex software systems, including compilers [33, 49] and the Linux kernel [50]. Specifically for SMT solvers, LLMs are being adapted to generate SMT-LIB inputs for fuzzing SMT solvers [38, 48]. Concretely, LaST [38] further retrains LLMs to enhance their capability in generating complete test formulas for SMT solvers. However, this approach often yields a high rate of invalid formulas due to syntactic errors and fails to leverage the potent skeletons of historical bug-triggering formulas. To overcome these limitations, MadFuzz introduces a complementary design. Instead of retraining models, we use in-context learning to guide the LLM in completing the skeletons of historical bug-triggering formulas. This design leverages existing effective structural patterns while avoiding the cost associated with additional training. By combining the strengths of LLMs with these domain-specific skeletons, MadFuzz generates more targeted and reliable test formulas. Our results show that this approach advances the direction of LLM-integrated SMT solver fuzzing and provides clearer benefits than existing LLM-based techniques.

3 Approach

In this section, we provide an overview of our proposed approach, MadFuzz, followed by a detailed explanation. We begin by describing how MadFuzz generates test formulas using LLMs through zero-shot learning and few-shot learning strategies. Subsequently, we demonstrate how MadFuzz discovers bugs in SMT solvers through differential testing. Finally, we elaborate on the feedback mechanism of MadFuzz to optimize the prompts and select input skeleton via sampling, improving the efficiency of the fuzzing loop.

3.1 Overview

This study presents a novel approach called MadFuzz, which is depicted in Figure 1. MadFuzz collects historical bug-triggering formulas from the bug tracking systems of open-source SMT solvers, including Z3, cvc5, Bitwuzla, and others. Subsequently, MadFuzz generates a mutated version of the bug-triggering formulas by removing some atomic formulas or their constituent elements (e.g., operators, constants, and variables), and replacing them with `<placeholder>`. Then, MadFuzz constructs instructions based on the mutated bug-triggering formulas and prompts LLMs to generate new test formulas. To be specific, MadFuzz supports two strategies for adapting LLMs to SMT solver testing: zero-shot learning and few-shot learning. In zero-shot learning, MadFuzz constructs instructions based on a prompt template and the mutated bug-triggering formulas for LLMs to generate new test formulas iteratively, which expects LLMs to infill novel elements and produce mutants. For few-shot learning, MadFuzz provides the LLMs with several question-answer examples, i.e., mutated bug-triggering formulas with `<placeholder>` and the corresponding original formulas, to improve the performance of LLMs in generating test formulas. The intuition behind this is that LLMs typically benefit from question-answer examples provided in the prompt to generate more accurate and useful text [4]. After obtaining the output of LLMs, MadFuzz instantiates the formulas generated through different strategies to ensure their validity and completeness. Finally, MadFuzz uses differential testing to identify bugs in SMT solvers by comparing the results of multiple SMT solvers on the same test formulas and witnessing whether the solvers exhibit anomalous behaviors. Additionally, MadFuzz incorporates a feedback mechanism based on the solvers' behaviors to improve the efficiency of fuzzing by optimizing the prompt and prioritizing skeletons that are more likely to produce effective mutants.

3.2 Bug-triggering Formula Collection

To generate new test formulas, MadFuzz requires a collection of historical bug-triggering formulas as basic materials. Multiple well-known SMT solvers, such as Z3 and cvc5, are open-source software available on GitHub. Consequently, users and test teams can report bugs in the issue tracking systems of these solvers, providing an opportunity to collect historical bug-triggering formulas. A complete bug report should contain all the essential information required for developers to reproduce the bug, including the bug-triggering formula, solver version, and reproduction commands. Therefore, MadFuzz can acquire formulas from the bug tracking systems of SMT solvers automatically using the GitHub API. First, MadFuzz collects the issues that report soundness bugs, invalid model bugs, and crashes, which can be inferred by the titles of issues. Then, it gathers the formulas from the issue bodies or comments based on predefined matching rules, since most formulas are in the fenced code blocks of `Markdown`. To achieve this, it employs a lightweight regular expression matching technique to extract SMT formulas from the fenced code blocks of `Markdown` in issue bodies or comments. The extracted bug-triggering formulas serve as the necessary materials for MadFuzz to generate new test formulas. Finally, we collect 3,877 bug-triggering formulas from the bug tracking systems of Z3, cvc5, Bitwuzla, Yices2, and OpenSMT.

3.3 Skeleton Enumeration with LLMs

Algorithm 1 presents the pseudocode of MadFuzz. Given the collected bug-triggering formulas H, MadFuzz leverages LLMs for SMT solver fuzzing using two strategies, namely zero-shot learning and few-shot learning. To ensure flexibility, the algorithm is parameterized by user-defined inputs, specifically: the set of target SMT solvers (*Solvers*), the total time budget for the fuzzing campaign (T_{budget}), and the number of mutants to generate per iteration (N_{mutant}).

Zero-shot Learning. MadFuzz can use zero-shot learning to guide LLMs in generating new SMT-LIB formulas from existing bug-triggering formulas. The key step is to provide the LLM with a partially redacted version of a

Algorithm 1: MadFuzz's pseudocode

input : $Solvers$, H, T_{budget}, N_{mutant}
output : Detected bugs $Bugs$

Procedure MadFuzz($Solvers$, H)
  $ScoreMap \leftarrow$ InitScore(H)
  $f_1 \leftarrow$ RandomSelect(H)
  $t_{start} \leftarrow$ CurrentTime()
  // Main Fuzzing Loop: Run until time budget is exhausted
  while CurrentTime() $- t_{start} < T_{budget}$ **do**
    $k_1 \leftarrow$ Index(H, f_1)
    repeat
      $f_2 \leftarrow$ RandomSelect(H)
      $k_2 \leftarrow$ Index(H, f_2)
      // MH Sampling: Check acceptance condition
    until $random.uniform(0,1) \leq (1-p)^{k_2-k_1}$
    // Generate N_{mutant} variants for the selected seed
    $ResList \leftarrow [\,]$
    for $i \leftarrow 1$ **to** N_{mutant} **do**
      $Prompt \leftarrow$ GenPrompt(f_2, H, $ResList$)
      $Formula \leftarrow$ Query($Prompt$, LM_g)
      $Result \leftarrow$ Detect($Formula$, $Solvers$)
      if $Result \neq Correct$ **then**
        $Bugs \leftarrow$ Add($Formula$)
      $ResList \leftarrow$ Add($Result$)
      // Update the score of seed formulas
      $ScoreMap \leftarrow$ Update($ScoreMap$, $Result$, H)
      $H \leftarrow$ Sort($ScoreMap$)
    $f_1 \leftarrow f_2$
  return $Bugs$

Procedure GenPrompt(f, H, $ResList$)
  $skeleton \leftarrow$ RandomMutate(f)
  $T \leftarrow InitTemplate$
  if CheckErr($ResList$) **then**
    $T \leftarrow$ Refine(T, f, LM_c)
  if *Zero-shot learning* **then**
    $Prompt \leftarrow$ Construct(T, $skeleton$)
  if *Few-shot learning* **then**
    $formulas \leftarrow$ RandomSample(H)
    $exs \leftarrow$ [RandomMutate($formulas$), $formulas$]
    $Prompt \leftarrow$ Construct(T, $skeleton$, exs)
  return $Prompt$

formula, its *skeleton*, and rely on the LLM to produce a complete mutation. This approach uses the instruction-following strengths and internal knowledge of modern LLMs while avoiding the need for handcrafted grammar rules. To construct a skeleton, MadFuzz removes selected elements from the bug-triggering formula and replaces them with `<placeholder>` tokens (Line 24). These removed elements include atomic formulas and the operators, constants, and variables appearing in those atomic formulas. The degree of removal depends on the structure of the original formula. If the formula has n atomic formulas, MadFuzz randomly removes fewer than n of them to ensure that the remaining structure still provides meaningful guidance to the LLM. This process introduces variation while preserving enough context for the model to generate a coherent mutation. Once the skeleton is created, MadFuzz combines it with a predefined prompt template called *InitTemplate* (Line 25). This template instructs the LLM to fill in the placeholders and produce a new SMT-LIB formula. The template is designed to be concise yet informative, encouraging the LLM to introduce additional variables or constraints when appropriate, while avoiding semantic inconsistencies. The prompt used by default is:

Prompt

Please create a mutated SMT-LIB formula that modifies the following version and infills the `<placeholder>`*. Feel free to introduce additional variables and constraints as needed. The goal is to create a mutated formula that is grammatically correct and does not introduce any semantic errors.*

This template is the default choice unless some specific conditions are met, which will be further elaborated on later.

MadFuzz then constructs the final instruction by combining the prompt template with the skeleton (Line 29). Using this instruction, the LLM iteratively generates new test formulas. By default, the iteration proceeds for $N_{mutant} = 10$ steps. This default value is user-configurable for specialized testing scenarios.

Few-shot Learning. Few-shot learning extends the above process by providing the LLM with example pairs that illustrate the desired transformation. Each example consists of a skeleton (the "question") and the original complete formula (the "answer"). These examples help the LLM better infer how placeholders should be replaced in similar contexts. To construct these examples, MadFuzz randomly selects up to three bug-triggering formulas (Line 31). We cap this number at three because larger sets increase the prompt length substantially, risking truncation due to LLM context limits. For each selected formula, MadFuzz removes elements following the same procedure as in zero-shot learning and pairs the skeleton with the original formula (Line 32). These examples are then embedded into the prompt template, enabling the LLM to observe concrete transformations before generating new formulas. With the examples included, MadFuzz constructs the full instruction and prompts the LLM to iteratively generate formulas in the same manner as zero-shot learning. This mode is particularly effective when the target formulas exhibit recurring structural patterns, as the examples help the LLM mimic those patterns more faithfully.

Formula Instantiation. After the LLM generates candidate formulas, MadFuzz applies a lightweight post-processing step to ensure they are syntactically valid and suitable for cross-solver testing. This step is crucial because LLMs may occasionally introduce solver-specific commands or incomplete constructs. First, MadFuzz scans the generated formulas using simple pattern matching to identify commands that fall outside the SMT-LIB standard, such as solver-specific `set-option` directives. Although valid for some solvers, these commands hinder differential testing because they prevent uniform execution across solvers. MadFuzz therefore removes such commands to maintain cross-solver compatibility. Next, MadFuzz verifies the syntactic correctness of each formula using the SMT-LIB parser from cvc5. If the parser reports an error, MadFuzz feeds the error message back into the LLM and asks it to produce a corrected version. This repair step is applied only once per formula

to limit overhead while still providing an opportunity to fix common issues such as unmatched parentheses or incomplete declarations. By combining syntax cleanup, standardization, and limited self-correction, MadFuzz increases the likelihood that generated formulas are both valid and diverse. These measures ensure that the formulas are ready for differential testing across multiple SMT solvers, thereby improving the effectiveness and robustness of the fuzzing process.

3.4 Differential Testing

To uncover bugs in SMT solvers, we employ differential testing, which involves comparing the results of multiple SMT solvers on the same test formulas. Our approach, MadFuzz, systematically scrutinizes the results yielded by SMT solvers (Lines 15 - 17), considering various facets such as satisfiability, model correctness, and anomalous behaviors. Regarding satisfiability, we discern a potential soundness bug when one solver indicates `sat` for a given test formula, while another solver deems it `unsat`. To determine the responsible solver, we utilize the `get-model` command to retrieve the model (i.e., an assignment expected to satisfy the formula) of the test formula from the solver that renders `sat`. We then evaluate the test formula using the retrieved model and the solvers under test to identify the solver responsible for the bug. If the formula can be evaluated to `sat` using the model, we consider it a soundness bug in the solver that returns `unsat`, or vice versa. MadFuzz also ensures that solvers validate the correctness of the model provided by the solver that returns `sat`. If the formula cannot be evaluated to `sat` using the model, we classify it as an invalid model bug. Finally, if any solver exhibits anomalous behavior, such as assertion violations, the associated test formula will be recorded as a crash bug. After manual inspection, we report the identified bugs to the corresponding solver's bug tracker.

3.5 Feedback Mechanism

To enhance the efficiency of its fuzzing loop, MadFuzz incorporates a feedback mechanism that continuously adapts the fuzzing process based on solver responses observed during differential testing. Each generated formula is classified into one of three categories: *bug-triggering*, *invalid*, or *valid*. Bug-triggering formulas highlight cases where at least one solver behaves inconsistently, signaling potential defects or unimplemented corner cases. Invalid formulas contain syntactic or structural errors that prevent reliable execution and typically indicate weaknesses in the current prompt or seed configuration. Valid formulas, in contrast, are accepted by all solvers without discrepancies and serve as stable anchors for guiding subsequent generation. By monitoring the evolution of these categories over time, MadFuzz dynamically adjusts both its prompt design and seed selection strategies, ensuring that the fuzzing effort remains targeted and responsive to solver behavior. The feedback mechanism enhances test generation along two complementary dimensions: prompt optimization and skeleton selection. On the one hand, when MadFuzz identifies invalid formulas based on solvers' feedback, it uses this information to refine the prompt presented to the generation LLM, making the instructions more precise and better aligned with SMT-LIB constraints. On the other hand, when new bug-triggering formulas are detected, MadFuzz leverages an MCMC-based sampling strategy to prioritize formula patterns and seeds that are statistically more likely to trigger solver inconsistencies. This dual perspective, repairing weakness in generation while amplifying promising signals, enables MadFuzz to evolve throughout the fuzzing campaign.

Prompt Optimization. During fuzzing, MadFuzz logs solver outcomes for every generated formula and stores them in a list *ResList* (Line 18). This history enables MadFuzz to identify early signs of degraded generation quality. Specifically, if the most recent three generated formulas are all invalid, MadFuzz initiates the prompt refinement process (Lines 26–27). The optimization LLM, distinct from the generation LLM, is tasked with improving the most recent prompt that led to the generation of invalid test formulas. Here, we employ the optimization instructions proposed by the existing prompt optimization technique [57]. As a result, MadFuzz acquires an updated prompt template, which is expected to better direct the LLMs towards the creation of more efficacious test formulas [25].

MadFuzz will use the optimized prompt to direct the generation LLMs to produce test formulas subsequently. By continuously grounding prompt revision in solver-derived feedback and iteratively correcting deficiencies as they arise, MadFuzz maintains a robust, adaptive, and self-improving test generation pipeline throughout the fuzzing campaign.

Skeleton Selection. During the fuzzing process, MadFuzz maintains a score for each bug-triggering formula, initialized to 0 (Line 2). After each iteration, MadFuzz updates the score of the bug-triggering formula based on the behavior of the solvers (Line 19). Specifically, if the mutant of the bug-triggering formula triggers a bug, MadFuzz increases its score by 1. If the mutant is rejected by the solvers, MadFuzz decreases the bug-triggering formula's score by 0.1, while the score remains unchanged if the mutant is accepted by the solvers and does not trigger bugs. To prioritize bug-triggering formulas likely to generate valid and effective test cases, MadFuzz employs a sampler to guide the selection of formulas for mutation. This selection process is facilitated by the Metropolis-Hastings (MH) algorithm [9], an MCMC method designed for obtaining random samples from a specified probability distribution. The algorithm iteratively generates a sequence of samples approximating the desired distribution, with the acceptance probability (α) of a formula f_2 contingent on the current formula f_1. In our context, the desired distribution is modeled as a geometric distribution, representing the probability distribution of the number X of Bernoulli trials needed to achieve one success. The probability of the k^{th} trial being the first success is given by $Pr(X = k) = (1-p)^{k-1}p$ for $k = 1, 2, 3, ...$ Subsequently, bug-triggering formulas are arranged in a list H, ordered in descending order based on their scores. Specifically, the first bug-triggering formula in the list has the highest score, while the last has the lowest. The MCMC process ensures that the sampled formulas conform to the geometric distribution, making higher-scoring formulas more likely to be selected than lower-scoring ones.

The Metropolis-Hastings algorithm selects formulas randomly and determines acceptance or rejection through a Metropolis choice:

$$\alpha = min(1, \frac{Pr(f_2)\, g(f_2 \to f_1)}{Pr(f_1)\, g(f_1 \to f_2)})$$

where $g(f_1 \to f_2)$ is the proposal distribution describing the conditional probability of proposing a new sample f_2 given f_1. Since the formulas are selected randomly, the inherently symmetric proposal distribution simplifies the acceptance probability to:

$$\alpha = min(1, \frac{Pr(f_2)}{Pr(f_1)}) = min(1, (1-p)^{k_2 - k_1})$$

where k_1 and k_2 are the indices of f_1 and f_2 in H based on their scores. The significance of α lies in its role: if f_2 has a higher score than f_1, f_2 is always accepted; otherwise, the proposal is accepted with a certain probability. Furthermore, the bigger k_2 - k_1 is, the less the acceptance probability is. It is noteworthy that at each iteration, scores need to be recalculated, and formulas must be resorted (Lines 19 - 20).

Parameter estimation. The geometric distribution parameter p governs how strongly the sampler favors high-ranked formulas. Let n denote the number of formulas in H. Following established practice [8], we set p by enforcing two practical constraints. First, the distribution should place nearly all probability mass within the available range $[1, n]$, ensuring that sampling does not overemphasize out-of-range tail mass:

$$\sum_{k=1}^{n} \Pr(X = k) \approx 1.$$

Second, the top-ranked formula should be selected more frequently than under uniform sampling. This requires:

$$p = \Pr(X = 1) > \frac{1}{n}.$$

In our setting, the corpus contains $n = 3,877$ formulas. We set $p = 0.0005$, which satisfies $p > 1/n \approx 0.00026$, thereby ensuring that the highest-ranked formula is selected with nearly twice the probability of a uniform strategy. At the same time, this choice of p preserves a long, slowly decaying tail: even mid- and low-ranked formulas retain nontrivial probability of being selected. This balancing effect is essential in fuzzing, as it promotes exploration of diverse formulas while still prioritizing those with historically strong performance. A larger value of p would overly concentrate probability near the top of the ranking and risk premature convergence, whereas a much smaller value would yield insufficient prioritization. The selected value provides a practical compromise between exploitation and exploration.

3.6 Implementation

We have implemented MadFuzz as a practical tool for SMT solver fuzzing using Python. Although MadFuzz is a universal framework for fuzzing SMT solvers that can be implemented based on any instruction fine-tuned LLMs, we finally select CodeLlama-7b-Instruct [37] and Llama-2-7b [42] to serve as the generation LLM and optimization LLM, respectively. The reason is that they are advanced open-source LLMs, which exhibit superior performance in various tasks and understanding human instructions. CodeLlama demonstrates remarkable proficiency in programming tasks. In terms of test input generation configuration, our default parameters include a temperature setting of 0.7 [4, 20, 38], a maximum output length of 2,048 characters, and a top_p value of 0.95 [37]. For the experimental evaluation, we defined a standard fuzzing time budget (T_{budget}) of 24 hours per session, allowing for the inspection of discovered bugs prior to any subsequent fuzzing campaigns.

4 Experimental Evaluation

This section presents a comprehensive evaluation of the effectiveness of MadFuzz.

4.1 Evaluation Setup

Research Questions. The experiments conducted aim to answer the following research questions:

- **RQ1:** How effective is MadFuzz in terms of zero-shot learning and few-shot learning strategies? (Section 4.2)
- **RQ2:** How does MadFuzz compare with state-of-the-art SMT solver fuzzers? (Section 4.3)
- **RQ3:** How effective is MadFuzz's feedback mechanism? (Section 4.4)
- **RQ4:** Can MadFuzz be used to expose new real bugs in SMT solvers? (Section 4.5)

Types of Bugs. As described in Section 3.4, bugs in SMT solvers can be classified into three main different types. This categorization is consistent with previous studies [39, 46, 47, 52]. The three categories of bugs are defined as follows:

• *Soundness bugs*: This type of bug occurs when two solvers provide opposite results for the same formula, where one solver reports `sat` while the other reports `unsat`. Such inconsistencies are considered soundness bugs.
• *Invalid model bugs*: An invalid model refers to a solver correctly identifying a formula as satisfiable (`sat`), but the model provided by the solver is incapable of satisfying the constraints in the formula.
• *Crash bugs*: This type of bug occurs when a solver exhibits abnormal behavior during the solving process, such as assertion violations and segmentation faults.

Environment. We conduct all of our experiments on a machine equipped with an Intel Xeon CPU Gold-6248 processor (20 cores and 256GB RAM) and NVIDIA Tesla V100 GPU running the Ubuntu 20.04 64-bit operating

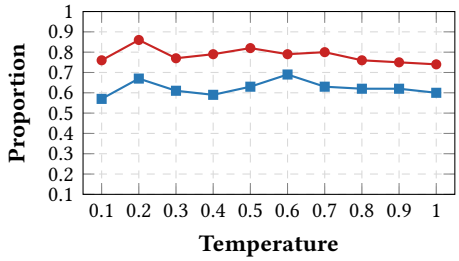

Fig. 2. Comparison of valid formula proportions generated by MadFuzz$_{ZS}$ (blue ■ for zero-shot) and MadFuzz$_{FS}$ (red • for few-shot) across various temperatures.

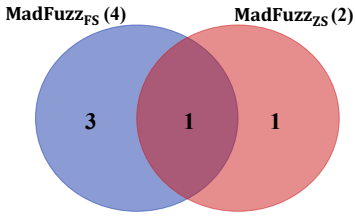

Fig. 3. The distribution of bugs detected by MadFuzz$_{ZS}$ and MadFuzz$_{FS}$ on previous versions of solvers.

system. In line with prior studies [39, 52], we set the time limit for solving queries for each test formula to 10 seconds.

4.2 RQ1: Comparison of Learning Paradigms

This RQ focuses on evaluating the effectiveness of the two strategies proposed in MadFuzz, i.e., zero-shot learning and few-shot learning, in terms of their ability to generate valid instances, achieve code coverage, and discover bugs. In this experiment, we select Z3 and cvc5 as the target solvers, since they support a wide range of logics and are also used in most of the related studies [39, 48]. Other solvers, such as Bitwuzla, support only a limited range of logics or provide too few released versions to enable meaningful longitudinal analysis. As a result, we do not include them in this RQ. To prevent data leakage, experiments are conducted on Z3-4.12.3 and cvc5-1.0.9, versions released after the LLMs. Note that the issues associated with the collected bug-triggering formulas are also resolved before the release of the solver versions.

4.2.1 *Validity of Generated Instances.*

Experiment Setting. In this experiment, a formula is considered valid if it can be solved by either Z3 or cvc5 without encountering any errors. To comprehensively evaluate the validity of the formulas generated by MadFuzz, we sample 100 formulas from the historical bug-triggering formulas at random. This sample size aligns with the scale commonly used in prior works [14, 38, 39]. From each sampled formula, we use both MadFuzz$_{ZS}$ (zero-shot) and MadFuzz$_{FS}$ (few-shot) to generate 1,000 mutated formulas under different temperature settings. Using 1,000 mutants per configuration ensures that we capture stable trends in validity rates while keeping the computational cost manageable. The temperature parameter controls the variability in the model's output, and exploring its effects is important for understanding the trade-off between diversity and correctness. Following common practice in the literature [14, 38], we vary the temperature from 0.1 to 1.0 in increments of 0.1, which provides fine-grained coverage of the model's generation behavior in both low- and high-creativity settings. Finally, for each temperature and learning mode, we compute the proportion of generated formulas that are valid according to Z3 or cvc5.

Result. As depicted in Figure 2, the proportion of valid formulas generated by MadFuzz$_{FS}$ is consistently higher than that of MadFuzz$_{ZS}$. Specifically, the proportion of valid formulas generated by MadFuzz$_{FS}$ exceeds 70% at all temperatures. However, the proportion of valid formulas generated by MadFuzz$_{ZS}$ remains around 60% at different temperatures. This result demonstrates the superior effectiveness of the few-shot learning strategy in generating valid formulas by learning from examples.

Table 1. Code coverage achieved by MadFuzz$_{ZS}$ and MadFuzz$_{FS}$. The highest values are highlighted.

Paradigm	Z3		cvc5	
	line	*function*	*line*	*function*
MadFuzz$_{ZS}$	27.1%	31.5%	30.5%	44.1%
MadFuzz$_{FS}$	**31.2%**†‡	**33.8%**†‡	**33.0%**†‡	**47.0%**†‡

Note: The highest value in each column is highlighted in bold.
† Statistically significant improvement ($p < 0.05$) over MadFuzz$_{ZS}$.
‡ Large effect size (Cliff's $\delta > 0.474$).

4.2.2 ***Code Coverage***.

Experiment Setting. To evaluate the code coverage achieved by MadFuzz, we use both MadFuzz$_{ZS}$ and MadFuzz$_{FS}$ to generate 1,000 formulas each. This scale provides enough inputs to observe consistent coverage trends while keeping the analysis tractable, and it follows the typical configuration used in earlier studies on SMT input generation [34, 38]. Unless otherwise stated, we set the temperature to 0.7, which is a commonly adopted default that balances diversity and stability in generated formulas [38]. We then feed the different groups of formulas generated by MadFuzz to Z3 and cvc5, and utilize the gcov[3] tool to collect the solvers' code coverage. Additionally, we repeat this experiment 10 times and calculate the average of code coverage to mitigate the influence of randomness, which is also consistent with prior studies [38].

Result. Table 1 delineates the code coverage attained by MadFuzz$_{ZS}$ and MadFuzz$_{FS}$, respectively. Note that discussions pertaining to the remaining rows of the table are reserved for subsequent subsections. We can observe that MadFuzz$_{FS}$ consistently outperforms MadFuzz$_{ZS}$ in terms of code coverage across both solvers. Specifically, MadFuzz$_{FS}$ achieves a line coverage of 31.2% in Z3 and 33.0% in cvc5, while MadFuzz$_{ZS}$ achieves 27.1% in Z3 and 30.5% in cvc5. To assess whether the advantages of MadFuzz$_{FS}$ are statistically reliable, we applied the Mann–Whitney U Test [26] to the results from ten independent runs. We find that all p-values fall below the 0.05 threshold, confirming the statistical significance of the improvements in code coverage achieved by MadFuzz$_{FS}$. We further employ Cliff's δ [36, 51] to evaluate the practical significance of the improvements. The calculated Cliff's δ values are all above 0.474, indicating that the improvements of MadFuzz$_{FS}$ are practically significant.

4.2.3 ***Bug Exposing***.

Experiment Setting. We evaluate the bug-finding capability of MadFuzz$_{ZS}$ and MadFuzz$_{FS}$ by running each fuzzer for 24 hours and measuring the number of unique *known* bugs it can rediscover. This duration follows the standard evaluation practice in prior work [38], providing sufficient time for each fuzzer to explore diverse behaviors while keeping the experiment practical to reproduce. To identify the unique bugs, we employed the *Correcting Commit* approach [7, 39]. Specifically, we feed a potential bug-triggering instance generated by a fuzzer to different commit versions of the solvers and check whether the bug was fixed in one commit. If a bug can be triggered before a commit while it cannot be triggered after the commit, we consider this commit as the correcting commit of the bug. The bugs corresponding to different correcting commits are considered as different

[3]https://gcc.gnu.org/onlinedocs/gcc/Gcov.html

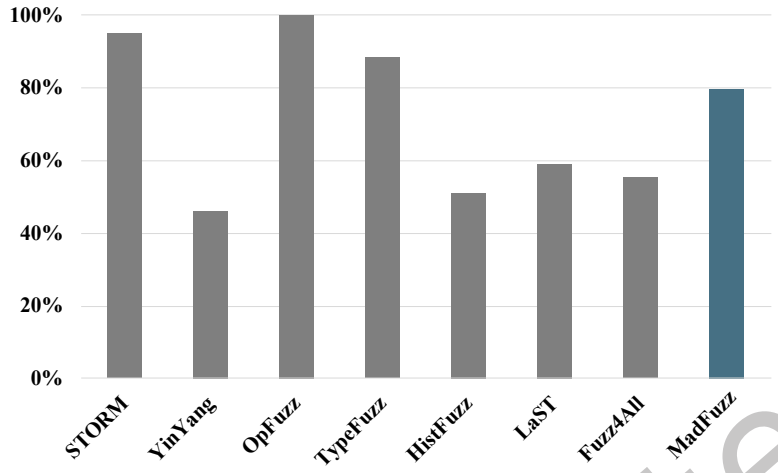

Fig. 4. Comparison of valid formula proportions generated by different SMT solver fuzzers.

bugs. In practice, we exploit binary search to accelerate the process of identifying correcting commits. Note that we focus on using fuzzing tools to find known bugs that have already been resolved, rather than finding new bugs in this experiment, which enables us to conveniently determine the number of unique bugs.

Result. As shown in Figure 3, MadFuzz$_{FS}$ detected four unique bugs in the solvers, while MadFuzz$_{ZS}$ detected two unique bugs. Concretely, MadFuzz$_{FS}$ was capable of detecting three bugs that MadFuzz$_{ZS}$ missed, while MadFuzz$_{ZS}$ identified one bug that MadFuzz$_{FS}$ failed to detect. Collectively, these results underscore MadFuzz$_{FS}$'s superior efficacy in generating valid formulas, achieving higher code coverage, and detecting bugs. Nonetheless, MadFuzz$_{ZS}$ demonstrates its value by complementing MadFuzz$_{FS}$ in bug detection capabilities.

4.3 RQ2: Comparison with Existing Fuzzers

To comprehensively evaluate the effectiveness of MadFuzz, we compare it with the state-of-the-art fuzzers, including STORM, YinYang, OpFuzz, TypeFuzz, HistFuzz, LaST, and Fuzz4All. They are all open-source fuzzers that can support multiple logics and detect a large number of bugs in SMT solvers. We use the latest versions of these fuzzers provided by their developers in our experiments. In this RQ, we also investigate the proportion of valid formulas, the achieved code coverage, and the number of unique bugs detected by different fuzzers. Note that, in the subsequent discussion, we use MadFuzz$_{FS}$ as the representative of MadFuzz in the comparison, since it outperforms MadFuzz$_{ZS}$ as illustrated in Section 4.2.

Validity of Generated Instances. To examine the validity of the formulas generated by different fuzzers, we employ the same experiment setting as in RQ1. Specifically, we employ the eight fuzzers to generate 1,000 formulas based on the 100 sampled formulas, respectively, and use Z3 and cvc5 to check whether the formulas are valid.

Figure 4 illustrates the proportion of valid formulas generated by different fuzzers. We can observe that the proportion of valid formulas generated by MadFuzz is higher than that of YinYang, HistFuzz, LaST, and Fuzz4All, while lower than that of STORM, OpFuzz, and TypeFuzz. The rationale behind this discrepancy lies in the fact that STORM, OpFuzz, and TypeFuzz primarily mutate operators in the formulas, resulting in a more restricted

Table 2. Code coverage and unique bugs detected. The highest values are highlighted. Statistical significance is calculated comparing MadFuzz against the second-best performing baseline.

Technique	Z3		cvc5		#Bug
	line	*function*	*line*	*function*	*Total*
STORM	15.3%	15.3%	21.0%	33.4%	0
YinYang	16.2%	16.3%	17.7%	31.0%	0
OpFuzz	20.4%	19.4%	23.0%	35.3%	2
TypeFuzz	22.9%	21.3%	24.4%	36.2%	1
HistFuzz	30.1%	31.2%	27.9%	43.4%	1
LaST	28.4%	26.9%	26.7%	40.8%	0
Fuzz4All	30.8%	33.2%	31.8%	45.8%	0
MadFuzz	**31.2%**†‡	**33.8%**†‡	**33.0%**†‡	**47.0%**†‡	**4**

Note: The highest value in each column is highlighted in bold.

† Statistically significant improvement ($p < 0.05$, Mann-Whitney U Test) compared to the best baseline.

‡ Large effect size (Cliff's $\delta > 0.474$).

mutation space and, consequently, a higher proportion of valid formulas. However, the restricted mutation space may also constrain the generation of diverse formulas and limit the exploration of solvers' code.

Code Coverage. To assess the code coverage achieved by each fuzzer, we employ the sampled 100 formulas in RQ1 as seeds and utilize the baseline fuzzers to generate 1,000 test formulas, respectively. These formulas were subsequently fed into the solvers, with the gcov tool being utilized to measure the resultant code coverage. This procedure was replicated 10 times as in RQ1.

Table 2 showcases the code coverage attained by test instances generated via different fuzzing techniques. Overall, we can observe that MadFuzz achieves superior code coverage across both solvers when compared to its counterparts. To ascertain the statistical significance of MadFuzz's performance over the second-best performing baseline (i.e., Fuzz4All), we also conducted the Mann-Whitney U Test and calculated the Cliff's δ values. The analysis confirmed the statistical and practical significance of the observed improvements. Therefore, we deduce that MadFuzz effectively improves the code coverage of SMT solvers compared to the existing SMT solver fuzzing techniques.

Bug-Finding Capability. Following the experimental setting in RQ1, we re-run each baseline fuzzer for 24 hours, recording the unique known bugs identified by each. The input seeds for these fuzzers are also the collected bug-triggering formulas. The findings, presented in Table 2, reveal that MadFuzz excels in detecting the highest number of unique bugs across the solvers. Furthermore, an interesting aspect of complementarity in bug detection emerged among OpFuzz, TypeFuzz, HistFuzz, and MadFuzz, with each identifying distinct bugs not found by the others. Note that the number of bugs discovered by the existing fuzzers was low, likely due to the resolution of the bugs reported by these fuzzers in the recent solver versions.

In conclusion, the experiments highlight MadFuzz's advantage in both bug detection and code coverage efficacy.

Table 3. Comparison of MadFuzz and MadFuzz$_{woFB}$ in terms of valid formula generation, code coverage achieved, and known bug detection.

Variant	Valid	Z3		cvc5		#Bug
		line	*function*	*line*	*function*	
MadFuzz	**79.6%**	**31.2%**	**33.8%**	**33.0%**	**47.0%**	**4**
MadFuzz$_{woFB}$	73.5%	30.1%	32.7%	30.4%	44.7%	2

Note: The highest value in each column is highlighted in bold.

4.4 RQ3: Efficacy of the Feedback Mechanism

This subsection delves into the efficacy of the innovative feedback mechanism implemented in MadFuzz through an ablation study. Here, we also employ MadFuzz$_{FS}$ as the representative of MadFuzz. We introduce MadFuzz$_{woFB}$, a variant of MadFuzz$_{FS}$ devoid of the feedback mechanism, to isolate and assess its contribution. Mirroring the experimental setup of RQ2, we evaluate the variant's performance in generating valid formulas, achieving code coverage, and detecting bugs.

Table 3 summarizes the results of the ablation study. First, we can observe that MadFuzz$_{woFB}$ can generate valid formulas at a rate of 73.5%, trailing behind MadFuzz$_{FS}$. Furthermore, MadFuzz$_{woFB}$ achieves 30.1% and 30.4% line coverage in Z3 and cvc5, respectively, which also falls short of those achieved by MadFuzz. Our statistical analysis confirms that these decreases in generation validity and code coverage are both statistically and practically significant. The feedback mechanism's importance is also apparent in bug detection. Although MadFuzz$_{woFB}$ identified two unique bugs, one of which was missed by the full MadFuzz system, its overall bug-finding effectiveness was lower. Ultimately, MadFuzz found a greater total number of unique bugs. Therefore, we can deduce that the feedback mechanism in MadFuzz is crucial for enhancing its performance in valid formula generation, code coverage, and bug detection.

4.5 RQ4: Bug Detection

The objective of *RQ3* is to examine whether MadFuzz can detect new real bugs in SMT solvers.

4.5.1 ***Experimental Setup.***

Given the high manual effort required for bug analysis, we focus this RQ on the few-shot learning strategy of MadFuzz, which demonstrated superior performance in RQ1, to detect new bugs in the SMT solvers.

Targeted Solvers. We select Z3, cvc5, and Bitwuzla as the target solvers in our experiments. Among them, Z3 and cvc5 are the two most widely used SMT solvers in the community. They support a comprehensive range of functionalities defined in the SMT-LIB standard and have been thoroughly validated by prior studies [34, 39, 46]. Bitwuzla is a relatively new advanced SMT solver. Although it only supports a narrow range of logics, "*Bitwuzla is a particularly good candidate since it is very robust*" as recommended by one of the cvc5 developers [38]. As such, we also include Bitwuzla in our experiments. Overall, the three solvers are representative of the leading SMT solvers, and it is challenging to expose bugs in them. To avoid duplicate bug reports, we always use MadFuzz to test the latest trunk version of the solvers.

Bug Inspection and Reduction. We inspected potential bugs identified by MadFuzz before reporting to prevent redundant reports. Specifically, we employ a bug grouping strategy for crashes, treating those affecting the same code line as identical, and use exception information from solvers for de-duplication. For soundness and invalid model bugs, we adopt a strategy similar to prior work [47], reporting reduced instances by theory categories

Table 4. Status of bugs found in the solvers.

Status	Z3	cvc5	Bitwuzla	Total
Reported	11	3	8	22
Confirmed	10	3	7	20
Fixed	10	2	7	19

Table 5. Bug types among the reported bugs.

Type	Z3	cvc5	Bitwuzla	Total
Crash	8	3	8	19
Invalid model	2	0	0	2
Soundness	1	0	0	1

and prioritizing unique bugs for initial reporting. Unaddressed reports are consolidated to minimize duplicates and ease developers' workload. Bug-triggering formula reduction is achieved using delta debugging tools like ddSMT [22] and, for non-compatible instances, C-reduce [35], facilitating developer understanding and resolution.

4.5.2 ***Results***.

Table 4 presents the status of the bugs discovered by MadFuzz in Z3, cvc5, and Bitwuzla. In total, MadFuzz has reported 22 bugs across the three solvers, of which 20 bugs have been confirmed by the developers. As of the time of writing, 19 bugs have been fixed, leaving the majority of the remaining bugs awaiting processing by the developers. It is worth noting that none of the bugs reported by MadFuzz have been marked as *won't fix* or *duplicate* by the developers, indicating that the bugs identified by MadFuzz are mostly authentic and that our inspection process before reporting is effective.

Table 5 depicts the distribution of the bug types of the reported bugs. Specifically, the majority of the reported bugs (19 out of 22) are crash bugs. The remaining bugs, including two invalid model bugs and one soundness bug, affect the correctness of the solvers. These results demonstrate that MadFuzz is capable of exposing real bugs in SMT solvers, and we deduce that it is more effective in detecting crashes compared to soundness and invalid model bugs.

5 Discussion

In this section, we provide an in-depth discussion of MadFuzz, addressing potential threats to its validity and exploring opportunities for future work.

5.1 Analysis on Generation

To explore the extent of mutations performed by LLMs, we analyze the differences between the mutants and their corresponding seeds in the fuzzing loop. Specifically, we investigate the proportion of distinct s-expressions (i.e., symbolic expressions) introduced by the LLMs in the mutants, compared to those in the seeds. S-expressions, commonly used in the SMT-LIB standard [2], are a notation for nested list data structures. For example, the term `(= (+ x 5) 10)` contains four s-expressions, excluding the term itself: `(+ x 5)`, `10`, `x`, and `5`. We randomly sample 1,000 seeds from the bug-triggering formulas, as well as the mutants generated based on them, and examine the differences in s-expressions between the mutants and their corresponding seeds. Specifically, we calculate the average ratio of different s-expressions in the mutants compared to the seeds at each iteration step. Figure 5 depicts the trend across each step. We observe that the difference in mutants continuously increases at each step, ultimately reaching a factor of three at the final step. This indicates that MadFuzz introduces many new s-expressions into the seeds, leading to extremely diverse mutants. Therefore, we conclude that LLMs are capable of effectively generating novel test formulas with high diversity.

In addition to diversity, we also examine the computational cost introduced by LLMs. We measure the time spent in different stages of MadFuzz, finding that generating a single mutant requires 2.1 seconds, which is

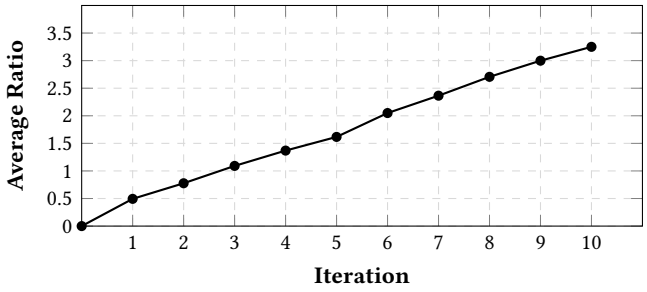

Fig. 5. Average ratio of distinct s-expressions between mutants and seeds in each iteration.

Table 6. Comparison of MadFuzz, MadFuzz$_{GPT}$, MadFuzz$_{DEEPSEEK}$, and MadFuzz$_{GEMINI}$ in terms of valid formula generation, code coverage achieved, and known bug detection.

Variant	Valid	Z3		cvc5		#Bug
		line	*function*	*line*	*function*	
MadFuzz	79.6%	31.2%	33.8%	33.0%	47.0%	4
MadFuzz$_{GPT}$	71.6%	31.5%	34.2%	33.1%	47.0%	5
MadFuzz$_{DEEPSEEK}$	75.8%	31.3%	35.1%	33.0%	47.4%	4
MadFuzz$_{GEMINI}$	86.1%	31.4%	34.4%	33.0%	47.7%	6

higher than conventional approaches. However, as demonstrated in Section 4.3 (*RQ2*), MadFuzz achieves superior bug detection within the same overall time budget. Moreover, this overhead can also be reduced through parallel execution on multiple GPUs or future advances in model design and computing resources. Hence, although the overhead is nontrivial, we consider it acceptable given the effectiveness of the approach.

5.2 Influence of Models

To evaluate the generality of MadFuzz across different LLMs, we implemented three additional variants using GPT-4 [32], DeepSeek-V3.2 [12], and Gemini-2.5 Flash [10], alongside the default CodeLlama-7b-Instruct and Llama-2-7b models. These models represent some of the most advanced LLMs currently available and have been widely adopted in recent research [16, 56]. We denote the corresponding variants as MadFuzz$_{GPT}$, MadFuzz$_{DEEPSEEK}$, and MadFuzz$_{GEMINI}$. For each variant, we measured three aspects: the proportion of valid test inputs generated, the code coverage achieved, and the number of unique bugs detected. The experimental setup follows the methodology described in Section 4, and the results are summarized in Table 6. Among the three variants, MadFuzz$_{GEMINI}$ produced the highest proportion of valid SMT formulas. However, code coverage and the number of detected bugs were comparable across all variants, and the observed differences in coverage were not statistically significant. Overall, these results suggest that the effectiveness of MadFuzz comes mainly from the proposed framework itself, rather than from the specific capabilities of the underlying LLMs.

5.3 Threats to Validity

This investigation is susceptible to several primary threats that warrant consideration.

Internal. The main internal threat is that the presence of randomness in the fuzzing process could introduce variability in the results of our experiments. To address this concern, we executed each experiment repeatedly and

documented the average results, adhering to norms in prior research [27, 38]. Besides, another threat arises from instances where MadFuzz failed to collect certain bug-triggering formulas from the issue trackers. As previously highlighted, a majority of the reports conform to standard and easily accessible formats. However, some reports pose challenges in accessibility. For instance, certain issues may encompass multiple zip files, complicating the determination of whether these files contain the pertinent formulas. It is noteworthy that such cases are infrequent.

External. The main external threat is that LLMs may "hallucinate", which means that they may produce inaccurate information. Furthermore, MadFuzz cannot guarantee the generation of valid formulas, which has the potential to impact the efficacy of testing procedures. However, according to statements provided by the developers of cvc5 [30], even invalid inputs are useful as they serve the purpose of validating the solvers' capability to handle such inputs accurately. Consistent with this view, we retain invalid formulas during fuzzing rather than filtering them out. This decision is further supported by our empirical findings: such cases have helped uncover bugs in parsing components, such as cvc5#9645.

5.4 Future Work

In this study, we mainly targeted SMT solver fuzzing. However, there are many testing techniques designed for other software systems based on the methods like skeleton enumeration. For example, Skeletal Program Enumeration (SPE) [55] is a technique for testing compilers, and Jattack [54] is a skeleton-based fuzzer, which utilizes manually written program templates to generate test cases for Java Virtual Machine (JVM). We believe that LLMs can revitalize these skeleton-based testing techniques. Thus, the integration of LLMs with these techniques is also worth investigating. Besides, MadFuzz is implemented based on CodeLlama-7b-Instruct [37] and Llama-2-7b [42], with a variant based on GPT-4 [31]. However, they are not the most advanced models available. Therefore, we believe it would be interesting to investigate whether the scale of models will affect the effectiveness of testing tools and how to select the most suitable model. Additionally, LLMs still have the potential to be involved in more stages of software testing. For example, we can utilize the code-understanding capability of LLMs to facilitate targeted fuzzing, further boosting the testing effect. We plan to explore these aspects in the future.

6 Conclusion

This paper introduces MadFuzz, an enhanced approach for testing SMT solvers that leverages the capabilities of LLMs to complete skeletons extracted from historical bug-triggering formulas. Specifically, we utilize zero-shot and few-shot learning paradigms to adapt LLMs to SMT solver fuzzing and incorporate a feedback mechanism to improve the efficacy of our approach. We implemented MadFuzz as a practical fuzzing tool and conducted a comprehensive evaluation using three SMT solvers, namely Z3, cvc5, and Bitwuzla. Notably, MadFuzz has successfully identified 20 confirmed real bugs in the solvers, with 19 of them already addressed by developers. Moreover, our experiments demonstrate that MadFuzz outperforms state-of-the-art SMT solver fuzzers in terms of code coverage and bug-finding ability. This study establishes the efficacy of LLMs as skeleton enumerators for SMT solver fuzzing, offering insights that could potentially inspire future research in other software system testing domains.

Acknowledgment

We sincerely thank the anonymous reviewers and editors for their constructive feedback and careful guidance, which substantially improved the quality and clarity of this work. We are also grateful to the Z3, cvc5, and Bitwuzla development teams for their generous support and for carefully reviewing the bugs we reported. Their insights and engagement were invaluable to this study. This work was supported in part by the National Natural

Science Foundation of China (Grants 624B2067, 62572226, 62372193, U2436207, and 62472215), the Jiangsu Natural Science Foundation (Grant BK20231402), and the Collaborative Innovation Center of Novel Software Technology and Industrialization. Additionally, we thank Yongcong Wang and Professor Hai Jin for their contributions to the conference version of this work.

References

[1] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 13243)*, Dana Fisman and Grigore Rosu (Eds.). Springer, 415–442. doi:10.1007/978-3-030-99524-9_24

[2] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2022. *The satisfiability modulo theories library (SMT-LIB).* Retrieved 2022-08-18 from http://smtlib.cs.uiowa.edu/

[3] Clark Barrett and Cesare Tinelli. 2018. Satisfiability modulo theories. In *Handbook of model checking*. Springer, 305–343.

[4] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.). https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html

[5] Robert Brummayer and Armin Biere. 2009. Fuzzing and Delta-Debugging SMT Solvers. In *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories* (Montreal, Canada) *(SMT '09)*. Association for Computing Machinery, New York, NY, USA, 1–5. doi:10.1145/1670412.1670413

[6] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs.. In *OSDI*, Vol. 8. 209–224.

[7] Junjie Chen, Wenxiang Hu, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. 2016. An empirical comparison of compiler testing techniques. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, Laura K. Dillon, Willem Visser, and Laurie A. Williams (Eds.). ACM, 180–190. doi:10.1145/2884781.2884878

[8] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. 2016. Coverage-directed differential testing of JVM implementations. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, Chandra Krintz and Emery D. Berger (Eds.). ACM, 85–99. doi:10.1145/2908080.2908095

[9] Siddhartha Chib and Edward Greenberg. 1995. Understanding the metropolis-hastings algorithm. *The american statistician* 49, 4 (1995), 327–335.

[10] Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit S. Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, Luke Marris, Sam Petulla, Colin Gaffney, Asaf Aharoni, Nathan Lintz, Tiago Cardal Pais, Henrik Jacobsson, Idan Szpektor, Nan-Jiang Jiang, Krishna Haridasan, Ahmed Omran, Nikunj Saunshi, Dara Bahri, Gaurav Mishra, Eric Chu, Toby Boyd, Brad Hekman, Aaron Parisi, Chaoyi Zhang, Kornraphop Kawintiranon, Tania Bedrax-Weiss, Oliver Wang, Ya Xu, Ollie Purkiss, Uri Mendlovic, Ilaï Deutel, Nam Nguyen, Adam Langley, Flip Korn, Lucia Rossazza, Alexandre Ramé, Sagar Waghmare, Helen Miller, Nathan Byrd, Ashrith Sheshan, Raia Hadsell Sangnie Bhardwaj, Pawel Janus, Tero Rissa, Dan Horgan, Sharon Silver, Ayzaan Wahid, Sergey Brin, Yves Raimond, Klemen Kloboves, Cindy Wang, Nitesh Bharadwaj Gundavarapu, Ilia Shumailov, Bo Wang, Mantas Pajarskas, Joe Heyward, Martin Nikoltchev, Maciej Kula, Hao Zhou, Zachary Garrett, Sushant Kafle, Sercan Arik, Ankita Goel, Mingyao Yang, Jiho Park, Koji Kojima, Parsa Mahmoudieh, Koray Kavukcuoglu, Grace Chen, Doug Fritz, Anton Bulyenov, Sudeshna Roy, Dimitris Paparas, Hadar Shemtov, Bo-Juen Chen, Robin Strudel, David Reitter, Aurko Roy, Andrey Vlasov, Changwan Ryu, Chas Leichner, Haichuan Yang, Zelda Mariet, Denis Vnukov, Tim Sohn, Amy Stuart, Wei Liang, Minmin Chen, Praynaa Rawlani, Christy Koh, JD Co-Reyes, Guangda Lai, Praseem Banzal, Dimitrios Vytiniotis, Jieru Mei, and Mu Cai. 2025. Gemini 2.5: Pushing the Frontier with Advanced Reasoning, Multimodality, Long Context, and Next Generation Agentic Capabilities. *CoRR* abs/2507.06261 (2025).

[11] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[12] DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo

Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jiawei Wang, Jin Chen, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Litong Wang, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runxin Xu, Ruoyu Zhang, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Shuting Pan, T. Wang, Tao Yun, Tian Pei, Tianyu Sun, W. L. Xiao, and Wangding Zeng. 2024. DeepSeek-V3 Technical Report. *CoRR* abs/2412.19437 (2024).

[13] Renzo Degiovanni, Dalal Alrajeh, Nazareno Aguirre, and Sebastián Uchitel. 2014. Automated goal operationalisation based on interpolation and SAT solving. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, Pankaj Jalote, Lionel C. Briand, and André van der Hoek (Eds.). ACM, 129–139. doi:10.1145/2568225.2568323

[14] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2022. Fuzzing Deep-Learning Libraries via Large Language Models. *CoRR* abs/2212.14834 (2022). arXiv:2212.14834 doi:10.48550/arXiv.2212.14834

[15] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. 2023. Large Language Models are Edge-Case Fuzzers: Testing Deep Learning Libraries via FuzzGPT. *CoRR* abs/2304.02014 (2023). arXiv:2304.02014 doi:10.48550/arXiv.2304.02014

[16] Zehang Deng, Wanlun Ma, Qing-Long Han, Wei Zhou, Xiaogang Zhu, Sheng Wen, and Yang Xiang. 2025. Exploring DeepSeek: A Survey on Advances, Applications, Challenges and Future Directions. *IEEE CAA J. Autom. Sinica* 12, 5 (2025), 872–893. doi:10.1109/JAS.2025.125498

[17] HistFuzz. 2023. . Retrieved 2023-07-30 from https://github.com/CGCL-codes/HistFuzz

[18] Heqing Huang. 2023. *CVE-2020-19725*. Retrieved 2023-11-5 from https://nvd.nist.gov/vuln/detail/CVE-2020-19725#

[19] Eunsuk Kang, Stéphane Lafortune, and Stavros Tripakis. 2019. Automated Synthesis of Secure Platform Mappings. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 11561)*, Isil Dillig and Serdar Tasiran (Eds.). Springer, 219–237. doi:10.1007/978-3-030-25540-4_12

[20] Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large Language Models are Few-shot Testers: Exploring LLM-based General Bug Reproduction. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2312–2323. doi:10.1109/ICSE48619.2023.00194

[21] Jongwook Kim, Sunbeom So, and Hakjoo Oh. 2023. Diver: Oracle-Guided SMT Solver Testing with Unrestricted Random Mutations. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2224–2236. doi:10.1109/ICSE48619.2023.00187

[22] Gereon Kremer, Aina Niemetz, and Mathias Preiner. 2021. ddSMT 2.0: Better Delta Debugging for the SMT-LIBv2 Language and Friends. In *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12760)*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer, 231–242. doi:10.1007/978-3-030-81688-9_11

[23] Yi Li, Aws Albarghouthi, Zachary Kincaid, Arie Gurfinkel, and Marsha Chechik. 2014. Symbolic optimization with SMT solvers. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, Suresh Jagannathan and Peter Sewell (Eds.). ACM, 607–618. doi:10.1145/2535838.2535857

[24] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023. Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing. *ACM Comput. Surv.* 55, 9 (2023), 195:1–195:35. doi:10.1145/3560815

[25] Ruotian Ma, Xiaolei Wang, Xin Zhou, Jian Li, Nan Du, Tao Gui, Qi Zhang, and Xuanjing Huang. 2024. Are Large Language Models Good Prompt Optimizers? arXiv:2402.02101 [cs.CL]

[26] H. B. Mann and D. R. Whitney. 1947. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *The Annals of Mathematical Statistics* 18, 1 (1947), 50 – 60. doi:10.1214/aoms/1177730491

[27] Muhammad Numair Mansur, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. 2020. Detecting critical bugs in SMT solvers using blackbox mutational fuzzing. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 701–712. doi:10.1145/3368089.3409763

[28] Marcílio Mendonça, Andrzej Wasowski, and Krzysztof Czarnecki. 2009. SAT-based analysis of feature models is easy. In *Software Product Lines, 13th International Conference, SPLC 2009, San Francisco, California, USA, August 24-28, 2009, Proceedings (ACM International Conference Proceeding Series, Vol. 446)*, Dirk Muthig and John D. McGregor (Eds.). ACM, 231–240. https://dl.acm.org/citation.cfm?id=1753267

[29] Aina Niemetz and Mathias Preiner. 2023. Bitwuzla. In *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 13965)*, Constantin Enea and Akash Lal (Eds.). Springer, 3–17. doi:10.1007/978-3-031-37703-7_1

[30] Aina Niemetz, Mathias Preiner, and Clark W. Barrett. 2022. Murxla: A Modular and Highly Extensible API Fuzzer for SMT Solvers. In *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part II (Lecture Notes*

in Computer Science, Vol. 13372), Sharon Shoham and Yakir Vizel (Eds.). Springer, 92–106. doi:10.1007/978-3-031-13188-2_5

[31] OpenAI. 2023. *ChatGPT: Optimizing language models for dialogue.* Retrieved 2023-12-1 from https://openai.com/blog/chatgpt

[32] OpenAI. 2023. GPT-4 Technical Report. *CoRR* abs/2303.08774 (2023). arXiv:2303.08774 doi:10.48550/ARXIV.2303.08774

[33] Xianfei Ou, Cong Li, Yanyan Jiang, and Chang Xu. 2024. 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, Volume 4, ASPLOS 2024, Hilton La Jolla Torrey Pines, La Jolla, CA, USA, 27 April 2024 - 1 May 2024*, Rajiv Gupta, Nael B. Abu-Ghazaleh, Madan Musuvathi, and Dan Tsafrir (Eds.). ACM, 298–312. doi:10.1145/3622781.3674171

[34] Jiwon Park, Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2021. Generative type-aware mutation for testing SMT solvers. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–19. doi:10.1145/3485529

[35] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, Jan Vitek, Haibo Lin, and Frank Tip (Eds.). ACM, 335–346. doi:10.1145/2254064.2254104

[36] Jeanine Romano, Jeffrey D Kromrey, Jesse Coraggio, and Jeff Skowronek. 2006. Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen'sd for evaluating group differences on the NSSE and other surveys. In *annual meeting of the Florida Association of Institutional Research*, Vol. 177. 34.

[37] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code Llama: Open Foundation Models for Code. *CoRR* abs/2308.12950 (2023). arXiv:2308.12950 doi:10.48550/ARXIV.2308.12950

[38] Maolin Sun, Yibiao Yang, Yang Wang, Ming Wen, Haoxiang Jia, and Yuming Zhou. 2023. SMT Solver Validation Empowered by Large Pre-Trained Language Models. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 1288–1300. doi:10.1109/ASE56229.2023.00180

[39] Maolin Sun, Yibiao Yang, Ming Wen, Yongcong Wang, Yuming Zhou, and Hai Jin. 2023. Validating SMT Solvers via Skeleton Enumeration Empowered by Historical Bug-Triggering Inputs. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 69–81. doi:10.1109/ICSE48619.2023.00018

[40] Julian Thomé, Lwin Khin Shar, Domenico Bianculli, and Lionel C. Briand. 2017. Search-driven string constraint solving for vulnerability detection. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE / ACM, 198–208. doi:10.1109/ICSE.2017.26

[41] Palina Tolmach, Yi Li, Shangwei Lin, Yang Liu, and Zengxiang Li. 2022. A Survey of Smart Contract Formal Specification and Verification. *ACM Comput. Surv.* 54, 7 (2022), 148:1–148:38. doi:10.1145/3464421

[42] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton-Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurélien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. *CoRR* abs/2307.09288 (2023). arXiv:2307.09288 doi:10.48550/arXiv.2307.09288

[43] Rachel Tzoref and Orna Grumberg. 2006. Automatic Refinement and Vacuity Detection for Symbolic Trajectory Evaluation. In *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 4144)*, Thomas Ball and Robert B. Jones (Eds.). Springer, 190–204. doi:10.1007/11817963_20

[44] Deze Wang, Zhouyang Jia, Shanshan Li, Yue Yu, Yun Xiong, Wei Dong, and Xiangke Liao. 2022. Bridging Pre-trained Models and Downstream Tasks for Source Code Understanding. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 287–298. doi:10.1145/3510003.3510062

[45] Po-Wei Wang, Priya L. Donti, Bryan Wilder, and J. Zico Kolter. 2019. SATNet: Bridging deep learning and logical reasoning using a differentiable satisfiability solver. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, 6545–6554. http://proceedings.mlr.press/v97/wang19e.html

[46] Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020. On the unusual effectiveness of type-aware operator mutations for testing SMT solvers. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 193:1–193:25. doi:10.1145/3428261

[47] Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020. Validating SMT solvers via semantic fusion. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 718–730. doi:10.1145/3385412.3385985

[48] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4All: Universal Fuzzing with Large Language Models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 126:1–126:13. doi:10.1145/3597503.3639121

[49] Chenyuan Yang, Yinlin Deng, Runyu Lu, Jiayi Yao, Jiawei Liu, Reyhaneh Jabbarvand, and Lingming Zhang. 2024. WhiteFox: White-Box Compiler Fuzzing Empowered by Large Language Models. *Proc. ACM Program. Lang.* 8, OOPSLA2 (2024), 709–735. doi:10.1145/3689736

[50] Chenyuan Yang, Zijie Zhao, and Lingming Zhang. 2025. KernelGPT: Enhanced Kernel Fuzzing via Large Language Models. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2025, Rotterdam, Netherlands, 30 March 2025 - 3 April 2025*, Lieven Eeckhout, Georgios Smaragdakis, Katai Liang, Adrian Sampson, Martha A. Kim, and Christopher J. Rossbach (Eds.). ACM, 560–573. doi:10.1145/3676641.3716022

[51] Yibiao Yang, Yuming Zhou, Jinping Liu, Yangyang Zhao, Hongmin Lu, Lei Xu, Baowen Xu, and Hareton Leung. 2016. Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su (Eds.). ACM, 157–168. doi:10.1145/2950290.2950353

[52] Peisen Yao, Heqing Huang, Wensheng Tang, Qingkai Shi, Rongxin Wu, and Charles Zhang. 2021. Skeletal approximation enumeration for SMT solver testing. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta (Eds.). ACM, 1141–1153. doi:10.1145/3468264.3468540

[53] Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, Dingyi Fang, Xiaoyang Sun, Lizhong Bian, Haibo Wang, and Zheng Wang. 2021. Automated conformance testing for JavaScript engines via deep compiler fuzzing. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 435–450. doi:10.1145/3453483.3454054

[54] Zhiqiang Zang, Nathan Wiatrek, Milos Gligoric, and August Shi. 2022. Compiler Testing using Template Java Programs. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 23:1–23:13. doi:10.1145/3551349.3556958

[55] Qirun Zhang, Chengnian Sun, and Zhendong Su. 2017. Skeletal program enumeration for rigorous compiler testing. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 347–361. doi:10.1145/3062341.3062379

[56] Wei Zhou, Xiaogang Zhu, Qing-Long Han, Lin Li, Xiao Chen, Sheng Wen, and Yang Xiang. 2025. The Security of Using Large Language Models: A Survey with Emphasis on ChatGPT. *IEEE CAA J. Autom. Sinica* 12, 1 (2025), 1–26. doi:10.1109/JAS.2024.124983

[57] Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. 2023. Large Language Models are Human-Level Prompt Engineers. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net. https://openreview.net/pdf?id=92gvk82DE-

[58] Xiaogang Zhu, Wei Zhou, Qing-Long Han, Wanlun Ma, Sheng Wen, and Yang Xiang. 2025. When Software Security Meets Large Language Models: A Survey. *IEEE CAA J. Autom. Sinica* 12, 2 (2025), 317–334. doi:10.1109/JAS.2024.124971

Received 20 June 2025; revised 24 November 2025; accepted 29 January 2026