SMT Solver Validation Empowered by Large Pre-trained Language Models

Maolin Sun*‡, Yibiao Yang*§‖, Yang Wang*‡, Ming Wen†¶, Haoxiang Jia†¶, Yuming Zhou*§‖
*State Key Laboratory for Novel Software Technology, Nanjing University, China
†School of Cyber Science and Engineering, Huazhong University of Science and Technology, China
‡{602023330025,njuwy}@smail.nju.edu.cn, §{yangyibiao, zhouyuming}@nju.edu.cn, ¶{mwenaa, haoxiangjia}@hust.edu.cn

***Abstract*—SMT solvers are utilized to check the satisfiability of logic formulas and have been applied in various crucial domains, including software verification, test case generation, and program synthesis. However, bugs hidden in SMT solvers can lead to severe consequences, causing erroneous results in these domains. Therefore, ensuring the reliability and robustness of SMT solvers is of critical importance. Despite several testing approaches proposed for SMT solvers, generating effective test formulas to comprehensively test SMT solvers remains a challenge. To address this challenge, in this study, we propose to port large language models (LLMs) to generate SMT formulas for fuzzing solvers. Specifically, the study presents a novel *retrain-finetune* pipeline to unleash the potential of language models to generate effective SMT formulas and improve their generation performance through data augmentation. We implemented our approach as a practical fuzzing tool, named LAST, and then extensively tested the state-of-the-art SMT solvers, namely Z3, cvc5, and Bitwuzla. To date, LAST has successfully uncovered 65 genuine bugs for the solvers, of which 45 have been fixed by the developers.**

***Index Terms*—SMT solver, fuzzing, large language model, retrain-finetune, data augmentation**

I. Introduction

Satisfiability Modulo Theory (SMT) solvers are sophisticated automated theorem proving tools that can check the satisfiability of first-order logic formulas in specific theories, such as arithmetic, arrays, and bit vectors [1]. Nowadays, SMT solvers are more versatile and applied in various fields, including software verification [2], program synthesis [3], and program analysis [4]. Given the wide range of applications for SMT solvers, ensuring their quality is of paramount importance. Specifically, solvers' quality affects the reliability and efficiency of their intended application. For example, as reported in the literature [5], [6], Amazon invokes SMT solvers tens of millions of times per day for service access control. As a result, the reliability and performance of SMT solvers can directly affect the stability of Amazon's services. Consequently, rigorous testing of SMT solvers is essential to ensure their quality.

To this end, researchers have proposed various techniques to generate test inputs for identifying bugs in SMT solvers. These techniques can broadly be categorized into two main categories: *formula generation* and *formula mutation*. Formula generation techniques, such as FuzzSMT [7] and Murxla [8], typically generate test formulas from scratch using predefined strategies. However, the diversity of the generated formulas is limited as they heavily rely on the designed strategies. By contrast, formula mutation techniques transform existing formulas in different ways to generate new test instances. For example, YinYang [9] combines two formulas with the same satisfiability to generate more intricate formulas with known ground truth that can be used to validate solvers. STORM [10] fragments a seed formula into multiple sub-formulas and reassembles them to produce satisfiable mutants. Additionally, OpFuzz [11] mutates the operators in formulas, and TypeFuzz [12] further expands its mutation space by generating new formula snippets. More recently, HistFuzz [13] has been proposed to utilize the elements mined from historical bug-triggering formulas of solvers to produce effective mutants. These techniques aim to stress-test SMT solvers by focusing on different aspects to mutate formulas. However, a limitation of those techniques, as pointed out in the prior studies [6], [12], is the restricted mutation space, which may hinder their effectiveness. Moreover, HistFuzz only utilizes the mined information in a rudimentary way as there may be many valuable clues for testing that remain hidden within the bug-triggering formulas.

Unfortunately, designing effective methods to generate vast and diverse test formulas for stress-testing SMT solvers is a challenging task. Besides, even experienced SMT solver experts may struggle to fully explore the effective elements in historical bug-triggering formulas and solver-specific behaviors for different solvers. Therefore, relying solely on human-written test formulas or formulas generated by separate techniques is insufficient for thoroughly validating SMT solvers. These challenges pose significant hurdles to the thorough testing of SMT solvers. Encouragingly, large pre-trained language models (LLMs) have shown stunning performance on various natural language and programming language tasks (e.g., code understanding [14], [15], program repair [16], and compiler testing [17]), which points out a promising way to address the challenges of lacking effective test formulas for SMT solvers. Although SMT formulas possess a highly specialized syntax and strict rules designed to express complex logical constraints, which distinguishes them from other natural and programming languages, they exhibit a level of "naturalness" similar to other languages [18]. Therefore, LLMs are also expected to be effective in generating SMT formulas. However, there is currently no empirical evidence to support the effectiveness of LLMs in generating test formulas. Furthermore, prior research [19] has highlighted the crucial role of high-quality

‖Yibiao Yang and Yuming Zhou are the corresponding authors.

training data in successfully transferring pre-trained LLMs to domains other than their original ones. This also poses a significant challenge for SMT solvers, as acquiring high-quality training data for SMT formula generation is hardly feasible.

Approach. In this paper, we propose a novel approach empowered by pre-trained LLMs to tackle the challenges of generating effective test formulas for SMT solver fuzzing. Our proposed approach involves two key steps, i.e., ***retraining*** and ***finetuning***. First, we ***retrain*** the LLMs on a large corpus of SMT formulas to enable them to acquire SMT-specific domain knowledge. To achieve this goal, we collect training data from the SMT-LIB official benchmarks [20], which comprise a vast number of formulas that strictly adhere to the standard specification. By retraining on this data, the LLMs can generate diverse and valid SMT formulas. Second, we further ***finetune*** the LLMs on historical bug-triggering formulas, which are known to involve structures that are more likely to trigger bugs [13] and solver-specific behaviors. The finetuning process equips LLMs with additional domain knowledge, guiding them to generate more test formulas capable of triggering SMT solver bugs. The rationale behind this approach is that the LLMs can be retrained to generate a massive number of valid SMT formulas, and the finetuning process can further guide the LLMs to generate effective test formulas that have the potential to unearth SMT solver bugs.

As discussed previously, effectively transferring pre-trained LLMs for SMT solver testing necessitates acquiring high-quality and abundant training data in the form of SMT formulas, which constitutes a critical yet challenging task. Hence, we propose two novel mutation strategies for data augmentation during the retraining and finetuning processes. For retraining, we propose *diversity-oriented* mutation to generate different formulas from existing ones for retraining the LLMs. Specifically, we design two types of mutation strategies to mutate the formulas in different dimensions, including *term mutation* and *operator mutation*. The term mutation generates mutants by adding, deleting, or replacing terms in the given formulas, while the operator mutation generates mutants by changing the operators in formulas, including theory function symbols and logical connectives. By together using these two types of mutation strategies, we can generate a large number of diverse mutants that cover comprehensive solving functionalities in SMT solvers, which alleviates the lack of high-quality training data for LLMs.

For finetuning, we use *semantic-preserving* mutation to augment bug-triggering formulas. This mutation strategy aims to preserve the mutants' abilities to trigger bugs. The previous study [13] has found that most semantic-equivalent mutants of bug-triggering formulas can also trigger the original bugs. Therefore, we utilize the functionalities of Z3 [21], one of the most well-known SMT solvers, to conduct semantic-preserving transformations for obtaining the semantic-equivalent mutants of bug-triggering formulas. This allows us to generate sufficient formulas for finetuning the LLMs.Ultimately, our proposed approach will result in a powerful LLM that can generate effective and diverse test formulas for testing SMT solvers.

Results. We implement our approach as a practical testing tool for SMT solvers, called LAST[1], based on the GPT-2 [22], a well-known pre-trained language model. We evaluate the effectiveness of LAST on three SMT solvers, namely Z3 [21], cvc5 [23], and Bitwuzla [24]. Ultimately, a total of 65 bugs are reported for these three state-of-the-art SMT solvers, of which 45 are fixed by the developers. Furthermore, we have found that LAST is capable of generating a large proportion of valid formulas, indicating the effectiveness of LLMs in generating content beyond conventional natural language and programming language. Additionally, we have also observed that the test formulas generated by LAST attain high code coverage and demonstrate effective bug detection capabilities when compared to other SMT solver testing tools. In summary, our approach of retraining and then finetuning LLMs for SMT solver testing shows great potential.

In this paper, we make the following main contributions:

- ***Originality:*** We propose a novel *retrain-finetune* pipeline for retargeting LLMs to generate diverse formulas that can stress-test SMT solvers. To the best of our knowledge, we are the first to employ LLMs for SMT solver validation. Our work provides empirical evidence for the effectiveness of LLMs in this domain.
- ***Novelty:*** We introduce two novel strategies, namely, *diversity-oriented* mutation and *semantic-preserving* mutation, to augment the training data for retraining and finetuning LLMs. These strategies enrich and diversify the data, enabling LLMs to generate effective SMT formulas that make a significant contribution to the field.
- ***Significance:*** Our primary objective is to identify authentic bugs in SMT solvers. We demonstrate that LLMs can generate content beyond typical natural and programming languages, indicating their potential for future researches.
- ***Usefulness:*** We have reported 65 bugs for three advanced SMT solvers, namely, Z3, cvc5, and Bitwuzla, of which 45 have been fixed by developers. Based on our evaluation results, our proposed tool LAST also demonstrates complementarity to existing SMT solver fuzzing techniques.

Paper Organization. The rest of this paper is structured as follows. Section II illustrates the high-level idea of our approach. Section III formalizes our approach and describes the implementation of LAST. Next, we elaborate on our extensive evaluation in detail in Section IV. In Section V, we conduct more discussions on the results. Finally, we survey related work in Section VI, and the conclusion is in Section VII.

II. Background and Motivation

A. SMT-LIB Language

The SMT-LIB initiative was started in 2003 [25] and has produced several versions of the SMT-LIB standard, which specifies comprehensive syntax and semantics for logic formulas.

[1] **L**LM-b**A**sed **S**M**T** solver fuzzer = LAST

```
(declare-fun x () Bool)
(declare-fun y () Real)
(assert (forall ((z Real)) (= x (< z y))))
(check-sat)
```

Fig. 1: A formula instance represented in SMT-LIB format.

The SMT-LIB language [26] is one of the most representative input languages for SMT solvers, which is adopted by the majority of solvers. In the specification, there are various defined commands for different purposes. The most basic commands are `declare-fun`, `assert`, and `check-sat`. The `declare-fun` command is used to declare new symbols. The `assert` command in SMT-LIB will add a formula to the current assertion stack, which is a collection of formulas that represent the problem to be solved. The `check-sat` command queries the solver to decide on the satisfiability of the current assertion stack. Therefore, an SMT solver will return `sat` only if all the formulas in the assertion stack can be satisfied simultaneously. In contrast, if no assignment can satisfy all the assertions, that instance is `unsat`.

Operators and *terms* are the fundamental building blocks of an SMT formula. Concretely, the predefined function symbols in SMT-LIB (e.g., $+, -, *, /$) are regarded as *operators* in the prior study [11]. *Term* is a syntactic expression that represents a value of a certain sort (e.g., `Int` and `Bool`). Terms can be constants, variables, the application of a function symbol or binders (e.g., universal quantifier `forall` and existential quantifier `exists`) to one or more terms. Figure 1 is an example of an SMT-LIB instance. In this example, the `declare-fun` command declares the variable x as a Boolean variable and the variable y as a real variable. The `assert` command adds a term, i.e., `(forall ((z Real)) (= x (< z y)))`, to the current assertion stack, which is a quantified formula bound by the variable `z`. In this term, there are some nested subterms such as `(< z y)`, `y`, and so on. Additionally, the formula also contains operators such as `forall`, `=`, and `<`. It is worth noting that the formula is simplified from the instance generated by LAST that reveals a new bug[2] in cvc5, which has also been fixed by the developers.

B. Large Pre-trained Language Models

Large pre-trained language models (LLMs) are typically transformer-based neural networks [27] that are trained on massive amounts of text data without any supervision or task-specific objective. The underlying principle is that by learning to predict the next word in a sequence of words, LLMs can capture the general patterns and structures of natural language. LLMs can be further finetuned on smaller and more specific datasets for downstream tasks, such as text classification, summarization, and generation. Besides, LLMs can also be retrained on program datasets to learn the syntax and semantics of different programming languages [19]. Consequently, they have been successfully applied in code-related tasks such as code completion and code summarization [19]. In addition, LLMs also offer an opportunity for software testing as they are capable of generating massive test cases and have shown to be effective in fuzzing JavaScript engines [17] and deep learning libraries [28], [29].

However, generating SMT formulas remains a challenge for LLMs, as SMT-LIB is a formal logic language with a highly specialized syntax and strict rules designed to express complex logical constraints. Despite this, SMT-LIB still exhibits a certain degree of "naturalness", which makes it possible to be learned by LLMs. For example, SMT formulas can represent the constraints of a program, implying that these formulas are inherently connected to the program's semantics. Drawing from the "naturalness" observed in programming languages [18], we can speculate that SMT formulas also exhibit a degree of "naturalness", thus suggesting the adaptability of LLMs for generating SMT formulas. However, this claim still lacks empirical evidence. Furthermore, re-purposing LLMs to produce effective SMT formulas necessitates high-quality training data, which is not easily accessible. In this study, we aim to tackle these challenges and investigate the feasibility of training LLMs for the purpose of fuzzing SMT solvers.

C. Data Augmentation

Data augmentation is a widely used technique aimed at enhancing the diversity and size of a dataset by applying various transformations to the original data [19]. This technique has been demonstrated to be effective in enhancing the generalization ability and robustness of deep neural networks [30]. Initially, data augmentation techniques were utilized in the field of computer vision [31], specifically involving transformations such as flipping, rotating, and color modification of images. These improve the effectiveness of model training by promoting diversity in training data. More recently, the application of data augmentation techniques has expanded to other domains, including code data [32]. For instance, the existing study [19] employed data augmentation techniques through the use of code transformation tools [33]. These tools can increase the size of the dataset with known labels through a series of transformations that preserve the semantics of the original code. The resulting augmented dataset has been shown to improve model performance in source code understanding.

Motivated by these recent advancements, we propose the integration of data augmentation techniques for SMT formulas as a solution to address the challenge of limited high-quality training data. This step is essential for effectively utilizing LLMs in the context of SMT solver fuzzing. By enriching the training dataset, we aim to enhance the generalization capability of the resulting models, ultimately leading to enhanced performance in generating effective SMT formulas.

III. Approach

In this section, we first present an overview of our proposed technique, LAST, as depicted in Figure 2, followed by a detailed elaboration. We describe how LAST processes and augments training data to retrain an LLM for SMT solver

[2] https://github.com/cvc5/cvc5/issues/9640

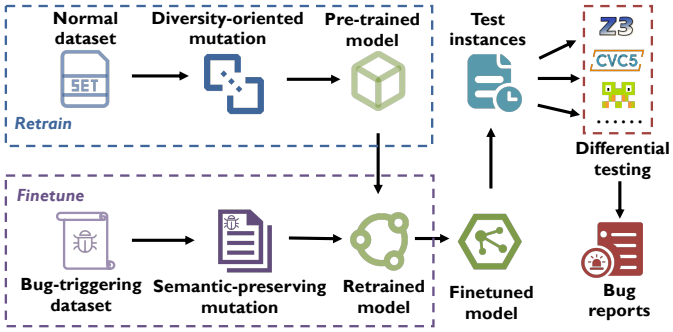


Fig. 2: Overview of LAST.

validation. Besides, we illustrate how LAST generates test instances to expose bugs in SMT solvers via differential testing.

A. Overview

In this study, we propose a *retrain-finetune* pipeline to learn SMT-specific knowledge and bug-triggering elements from normal formulas and bug-triggering formulas, respectively. With current LLMs typically embodying millions of parameters and further expected to grow in scale, it is crucial to gather a large amount of diverse training data to train LLMs for specific tasks [19]. Therefore, to address the challenge of the insufficient training dataset for retraining the LLMs and to strengthen them in generating bug-triggering formulas, we propose two tailored data augmentation strategies, i.e., diversity-oriented mutation strategy and semantic-preserving mutation strategy, to generate diverse and high-quality training data used for the retrain-finetune pipeline. Subsequently, we retrain the LLM with the formulas augmented by the diversity-oriented mutation. To strengthen the LLM to generate formulas that uncover bugs, we further finetune it by using the bug-triggering formulas along with the mutated ones from semantic-preserving mutation. Finally, we utilize the learned LLM to produce a huge amount of test formulas to fuzz SMT solvers. LAST spots bugs by seeking inconsistencies between multiple SMT solvers for solving the same formulas.

B. Data Collection

The success of transferring LLMs to generate formulas for testing SMT solvers largely depends on the diversity of training data used for retraining the LLMs. Therefore, the collection of high-quality training data is essential for effectively retraining the pre-trained LLMs. Fortunately, SMT-LIB offers official benchmarks [20] for SMT solvers, which contain hundreds of thousands of formulas that strictly adhere to the specifications. Typically, the majority of these formulas are derived from real-world applications or are used to evaluate performance, which are unlikely to trigger bugs in SMT solvers [13]. Additionally, the bug-tracking systems of open-source SMT solvers (e.g., Z3 and cvc5) often contain a lot of formulas that have triggered bugs in previous versions of the solvers. These formulas can be readily obtained from bug-tracking systems by retrieving the issue titles and extracting the formulas contained within the issues. Here, we employ the existing tool HistFuzz [34], which provides this service, to collect bug-triggering formulas. The formulas also include abundant solver-specific knowledge that complements the official benchmarks, which has not been fully explored in prior research. Previous studies [13], [35], [36], [37] have shown that these bug-triggering inputs are extremely effective in exposing bugs for SMT solvers and other software systems. Therefore, we first retrain the pre-trained language model by using the normal formulas from the official benchmarks and then finetune the retrained model with the bug-triggering formulas.

To ensure the training data is suitable for the LLM, we filter out excessively long formulas in the benchmarks [29] as they may hinder the LLM's ability to effectively understand and express complete semantics due to its limited context size (e.g., 1024 tokens for GPT-2 [38]). In addition, we observe that bug-triggering formulas are typically small in size, with over 90% of the formulas in these bug-triggering formulas having a size of less than 5 KB. Therefore, we also exclude formulas longer than 5 KB from the training data. After that, we gather 139,367 formulas as the training data for retraining the LLM, and 3,474 bug-triggering formulas originating from Z3 and cvc5 for finetuning it. Finally, we refactor the collected formulas to a uniform format before data augmentation. Concretely, to enhance the readability of formulas and prevent naming conflicts during data augmentation, we eliminate the `let` binders and rename variables based on their sorts. For instance, given the formula in Figure 1, we rename the variables `x` and `y` to `bool_0` and `real_0`, respectively.

C. Augmentation for Training Data

As our training dataset, especially the dataset for finetuning, is relatively small compared to existing studies [17], [19], we resort to data augmentation techniques to enhance the quantity and diversity of our training data. On the one hand, we apply diversity-oriented mutation to generate mutants from the formulas in the official benchmarks for retraining. On the other hand, we use semantic-preserving mutation on the bug-triggering formulas to generate mutants that retain their bug-triggering abilities as much as possible for finetuning.

1) Diversity-oriented Mutation:

To enhance the diversity of formulas, we introduce diversity-oriented mutation, which mutates the formulas from two perspectives, namely, term mutation and operator mutation.

For ***term mutation***, given a formula, we add, delete, or replace the terms in the formula to generate mutants.

Definition 3.1 (Term mutation): Let φ be an SMT formula, t_1 be a term in φ, and t_2 be an alternative term. Assuming t_1 and t_2 have the same sort, the *replace* term mutation mutates the formula φ by replacing the term t_1 with t_2, which can be expressed as $\varphi[t_2/t_1]$. The *add* term mutation mutates the formula φ by replacing the empty term with t_2, which can be expressed as $\varphi[t_2/\varnothing]$. The *delete* term mutation mutates the formula φ by replacing the term t_2 with the empty term, which can be expressed as $\varphi[\varnothing/t_1]$.

Algorithm 1 illustrates the process of term mutation. In the process, a formula f is initially selected for mutation, and a new

Algorithm 1: Term-level mutation

```
1  Function Term_level_mutate(f, c):
2      t ← RandomSelect(f)
3      mutator ← SelectMutator(t)
4      c' ← SortConversion(c)
5      if mutator = add then
6          mutant ← Add(f, t, c')
7      if mutator = delete then
8          mutant ← Delete(f, t)
9      if mutator = replace then
10         mutant ← Replace(f, t, c')
11     return mutant
```

term c is randomly chosen from the formulas in the benchmarks. The mutation process begins by randomly selecting a term t from f as the mutation target (Line 2). Next, the mutator (i.e., `add`, `delete`, or `replace`) is determined based on the operator that operates on t (Line 3). If the mutator's arity is not fixed (e.g., $+$, $-$, *and*, and *or*), a mutator is randomly selected from the set of `add`, `delete`, and `replace`. Otherwise, only the mutator `replace` is selected. To ensure that the mutant is well-typed, c must be converted to the sort of t if their sorts are different (Line 4), which is achieved by using the conversion functions defined in the SMT-LIB standard as shown in Table I. For example, if t is a real number and c is an integer, we can convert c to a real number by adding a `to_real` operator to it. With the help of these conversion functions, we can convert almost any term to a desired sort through their combination. For instance, if t is a real number and c is a string, we can first use the `str.to_int` operator to convert c to an integer and then use the `to_real` operator to convert the term to a real number. In practical implementations, we combine conversion functions when no function can directly convert the sort of the term c to the sort of the term t. If the term c cannot be converted to the sort of t, it will be discarded. Ultimately, the mutator is applied to f to produce a mutant (Lines 5-10), which is typically well-typed.

Operator mutation involves replacing the operators in a formula to produce mutants. To ensure producing well-typed mutants, only operators with consistent sorts of operands and return values as the original operator are considered as replacements. If multiple operators meet this requirement, one of them is selected at random. In this work, we adopt the method described in a prior study [11] to perform operator mutation. Our tool LAST randomly performs term mutation or operator mutation ten times to generate a mutant, and we produce ten mutants for each formula in the benchmarks. This results in over one million mutants in total.

2) Semantic-preserving Mutation:

Although term mutation and operator mutation can also enhance the diversity of mutants when applied to bug-triggering formulas, they may compromise the formulas' bug-triggering abilities and undermine our intention of finetuning. To mitigate this concern and ensure the preservation of bug-triggering abilities in the augmented data, we introduce semantic-preserving mutation. Specifically, we leverage the SMT solver Z3 to perform semantic-preserving mutation on the formulas by using its various built-in tactics[3], which can transform the original formulas into equivalent ones with identical semantics. A prior study [13] has demonstrated that the majority of mutants transformed by Z3's tactics can still trigger the same bugs as the original formulas. To ensure a comprehensive and non-redundant set of mutants, we filter out identical and duplicate mutants, resulting in a final collection of 19,062 mutants derived from the bug-triggering formulas.

TABLE I: Conversion function symbols defined in SMT-LIB.

Conversion Function Symbol	Description
`ite`	Convert `Bool` to any sort
`is_int`	Convert `Int` or `Real` to `Bool`
`to_real`	Convert `Int` to `Real`
`str.from_int,str.from_code`	Convert `Int` to `Str`
`int2bv`	Convert `Int` to `BitVec`
`to_int`	Convert `Real` to `Int`
`str.is_digit`	Convert `Str` to `Bool`
`str.to_int,str.to_code`	Convert `Str` to `Int`
`bv2nat`	Convert `BitVec` to `Int`

By employing the aforementioned data augmentation strategies, we generate a vast number of mutants from the benchmarks and bug-triggering formulas, thereby significantly enriching and diversifying the training data. This approach is expected to improve the overall performance of the LLM. With the augmented data, models can be trained to acquire abundant knowledge of SMT formulas, which aids in generating new test formulas for exposing bugs.

D. Test Formula Generation

The language model used in LAST is GPT-2 [22], a pre-trained model provided by OpenAI. This model has been trained on a vast corpus of natural language data. To adapt the model for generating SMT formulas, each formula in the augmented training dataset is represented as a sequence of numerical values. To achieve this, a vocabulary table is constructed, and each command, symbol, and constant in the SMT formulas is assigned an integer value. The vocabulary table is constructed using the Byte Pair Encoding (BPE) tokenization method [39], which is also utilized by GPT-2. BPE tokenization involves breaking down each word into subwords based on its frequency in the training data. Common words are tokenized as whole words (e.g., `and`, `or`, `not`), while infrequent words are broken down into characters that can be used to construct other words. This approach enables the representation of an infinite number of words in SMT formulas by reusing tokens from a finite list of subwords while minimizing the number of tokens required to represent the training dataset. Ultimately, the vocabulary table stores the integer representation of each subword or token.

Model Retraining. To expand GPT-2's ability to generate SMT formulas, a retraining process is undertaken. This involves

[3] https://microsoft.github.io/z3guide/docs/strategies/tactics

updating the weights of the pre-trained GPT-2 model using the formulas in the benchmarks, as well as their corresponding augmented versions through diversity-oriented mutation. The Adam optimizer [40] is used to train the network for 150,000 iterations, with an initial learning rate of 0.0001. In our practice, we retrain the model using two NVIDIA GeForce RTX 3080 Ti GPUs. It is crucial to note that the LLM is not designed for generating SMT formulas, and thus this retraining process is essential for this specific task.

Model Finetuning. After the retraining process, a finetuning phase is conducted to finetune the model using the bug-triggering formulas along with their corresponding mutants. This process aims to enhance the model's capacity by capturing bug-triggering elements and incorporating solver-specific knowledge. The finetuning process adopts the same training settings as the retraining process, with the exception that only the weights of the last two fully-connected layers in the model are updated. This modification is made due to the relatively small size of the training data, which is in line with a prior study [17]. Additionally, the training process is set to 50,000 iterations with the same initial learning rate.

Test Input Instantiation. Once the trained model is available, it can be utilized to generate new test formulas. Since the output of the model is a sequence of text that can contain multiple SMT formulas, it is necessary to instantiate the text into test instances. To achieve this, we first provide the model with a null prompt, which is essentially an empty string, and instruct it to generate a sequence of tokens in an auto-regressive manner. The null prompt is used because the customized model has demonstrated satisfactory performance for various tasks when provided with only a null prompt [41]. In practice, we query the model to generate ten samples at a time with a random seed at a temperature of 0.7. This temperature parameter controls the level of creativity in the model's output. Due to the limit of output tokens, which are 1024 for GPT-2, each generated formula is not guaranteed to be complete. Consequently, we need to identify the longest complete formula in the generated sequence and utilize it as a test input. Specifically, if the incomplete formula solely lacks closing parentheses, we append them to the end of the formula. Alternatively, if the assert statement lacks more than just parentheses, we discard it and use the complete assert statements above it in the generated formulas as the test input. It is crucial to note that this procedure is vital to ensure that the generated test inputs are complete. Identifying the longest complete formulas in the generated sequence increases the likelihood that the test instances will be executed normally by SMT solvers.

E. Differential Testing

To expose bugs in SMT solvers, we adopt differential testing, which involves comparing the results of multiple SMT solvers on the same test formulas. Specifically, LAST inspects the results of SMT solvers from various aspects such as satisfiability and model correctness. Regarding satisfiability, if a solver returns `sat` for a test formula but another solver returns `unsat`, we consider it a potential soundness bug. To determine which solver is responsible for the bug, we use the `get-model` command to retrieve the model (i.e., an assignment that is expected to satisfy the formula) of the test formula using the solver that returns `sat`. We then use the model to evaluate the test formula using the solvers under test to determine which solver is 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. To avoid false positives, we exclude cases that embody the under-specified behavior of SMT-LIB in different implementations of solvers. To identify invalid model bugs, LAST uses the model to evaluate the test formula when the solvers return `sat` correctly. If the formula cannot be evaluated to `sat` using the model, we consider it an invalid model bug. Finally, if any of the solvers exhibit abnormal behaviors (e.g., assertion violations), the corresponding test formula will be recorded as a crash bug. After a manual inspection, we report the identified bugs to the bug tracking systems of the corresponding solver.

IV. Evaluation

This section presents a comprehensive evaluation of the effectiveness of LAST. The experiments conducted aim to answer the following research questions:

- **RQ1:** Can LAST generate valid SMT formulas? (Section IV-A)
- **RQ2:** Can LAST be used to expose authentic bugs in SMT solvers? (Section IV-B)
- **RQ3:** Does LAST complement the state-of-the-art SMT fuzzers? (Section IV-C)
- **RQ4:** How effective are the major components of LAST? (Section IV-D)

Types of Bugs. As described in Section III-E, bugs in SMT solvers can be classified into three main different types. This categorization is consistent with previous studies [9], [11], [42], [6], [13]. 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 as 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-6230 processor (40 cores and 128GB RAM) running the Ubuntu 20.04 64-bit operating system. To facilitate identifying and de-duplicating crashes such as memory leaks and use-after-free bugs, we build Z3 and cvc5 with AddressSanitizer [43]. In line with prior studies [42], [13], we set the time limit for solving queries for each test formula to 10 seconds.

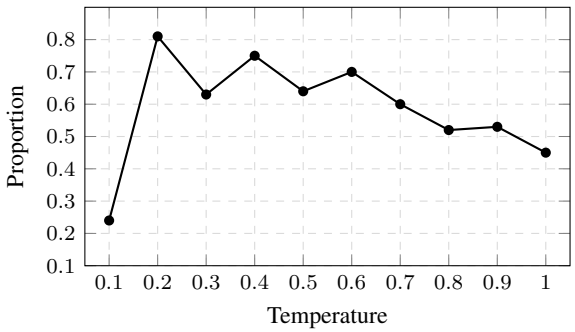


Fig. 3: Proportion of valid formulas generated by LAST at different temperatures.

A. RQ1: Validity of Generated Formulas

In *RQ1*, we aim to investigate the effectiveness of LAST in generating valid SMT formulas.

1) Experimental Setup:

In this experiment, a formula is considered valid if it can be solved by either Z3 or cvc5 without encountering any errors. Z3 or cvc5 are two established SMT solvers that support a wide range of logics and have been employed in numerous related studies for evaluation purposes [9], [11], [12], [13], [10]. To comprehensively evaluate the validity of the formulas generated by LAST, we utilize the model to create 1,000 formulas at different temperatures. The temperature is used to control the level of creativity in the model's output. We inspect these formulas with respect to different temperatures using Z3 and cvc5. Specifically, we generate 1,000 formulas at each of the temperatures from 0.1 to 1.0 with an interval of 0.1. This temperature range is commonly employed in prior studies [28]. Ultimately, we calculate the proportion of valid formulas generated by LAST at different temperatures.

2) Results:

The results of the experiment are presented in Figure 3. We can find that LAST produces a relatively high proportion of valid formulas when the temperature is set between 0.2 and 0.7, which is more than 60%. Notably, the highest proportion of valid formulas (81%) is generated at a temperature of 0.2. These results suggest that the trained model is capable of generating valid formulas with a high probability within appropriate temperature ranges. Therefore, to ensure LAST has a high level of creativity in generating formulas and to make it have a relatively high probability of generating valid formulas, we set the temperature to 0.7 in LAST.

B. RQ2: Bug Detection

The objective of *RQ2* is to examine whether LAST can detect authentic bugs in SMT solvers, which serves as an indicator of the effectiveness of LAST.

1) Experimental Setup:

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 [11], [12], [13]. Bitwuzla is a relatively new advanced SMT solver recommended by one of the cvc5 developers. Although it only supports a narrow range of logics, "*Bitwuzla is a particularly good candidate since it is very robust*" as the developer claims. As such, we also include Bitwuzla in our experiments to assess the effectiveness of LAST. Overall, the three solvers are representative of the state-of-the-art SMT solvers, and it is extremely challenging to expose bugs in them. In order to expose new bugs, we always use LAST to test the latest trunk version of the solvers.

Selected Options. Similar to prior studies [12], [13], our primary focus in this study is on detecting bugs in the default mode of the SMT solvers (i.e., no additional options are enabled). However, certain options are indispensable, such as the "model_validate=true" option in Z3, "–check-models" option in cvc5, and "–check-model" option in Bitwuzla, which are enabled to detect invalid model bugs. These options are considered to be part of the default mode of the solvers [11]. Furthermore, some important options deserve to be tested, e.g., Z3's new core option "tactic.default_tactic=smt, sat.euf=true", which is expected to become the default mode of Z3 as it matures [12]. Additionally, the developers of cvc5 provide a list of options that warrant testing and serve as proper guidelines for fuzzing [44]. Therefore, these options are also included in our experiments. However, since Bitwuzla does not support many options, we only test its default mode.

Bug Inspection and Reduction. Despite considering that false positives can be induced by the under-specified semantics of SMT-LIB, it remains imperative to thoroughly inspect the potential bugs identified by LAST to avoid duplicate reports. When crashes occur, we utilize a bug grouping approach to identify those crashes pointing to the same line of code as the same bug and subsequently search for exception information thrown by the solvers in the bug tracking systems for de-duplication. For soundness bugs and invalid model bugs, similar to prior work [9], we report one reduced instance at a time according to their theory categories. In cases where multiple formulas with the same theory reveal the same bug, we prioritize reporting one of them first. Subsequently, we check whether the remaining formulas can still trigger bugs, and determine whether or not to report them. If the report has not been addressed, a consolidated issue is created by grouping possible relevant bug reports, as suggested by a primary developer of Z3. This can help avoid duplicate reports and reduce the workload of developers. Moreover, we also need to reduce the complexity of the bug-triggering formulas to make developers easier to understand, confirm, and fix for developers. To achieve this, we use delta debugging tools specifically designed for the SMT-LIB language, e.g., ddSMT [45] and pyDelta [46], to reduce the formulas before reporting them.

2) Results:

Statistics of Bugs. Table II illustrates that LAST has identified 65 bugs in the SMT solvers, of which 45 have been fixed by the developers. The majority of the remaining bugs are currently awaiting processing by the developers. While we

TABLE II: Status of bugs found in Z3, cvc5, and Bitwuzla.

Status	Z3	cvc5	Bitwuzla	Total
Reported	37	24	4	65
Fixed	21	20	4	45
Duplicate	1	0	0	1
Won't fix	1	1	0	2

TABLE III: Bug types among the fixed bugs.

Type	Z3	cvc5	Bitwuzla	Total
Crash	13	19	3	35
Invalid model	6	1	1	8
Soundness	2	0	0	2

made a concerted effort to inspect and de-duplicate the bugs before reporting them, it is impossible to guarantee the complete absence of duplication. Nevertheless, the small number of duplicate bugs indicates that the de-duplication process is effective. In addition, our report includes 2 bugs that the developers do not plan to address. The main reasons are either the current lack of bandwidth to address them or the determination that the reported issues do not qualify as bugs.

Table III presents the bug types of fixed bugs. Notably, the majority of the fixed bugs (35 out of 45) are crash bugs, while the remaining bugs include eight invalid model bugs and two soundness bugs that affect the correctness of the solvers. Of the fixed bugs, most (34/45) are found in the default mode of the solvers, with the remainder discovered in the new core options of Z3 (six fixed bugs) and the options recommended for fuzzing by cvc5's developers (four fixed bugs). In summary, LAST is capable of identifying real bugs in SMT solvers.

Significance. To better understand the significance of the bugs identified by LAST, we investigate how many of them impact the release versions of Z3 and cvc5. This analysis sheds light on the lifespan of the bugs. It is important to note that we exclude the bugs identified in Bitwuzla from our analysis, as it is a newly developed SMT solver that does not yet have any other official release versions.

To investigate the longevity of bugs, we select Z3 release versions from 4.8.1 to 4.11.0 and cvc5 release versions from 0.0.2 to 1.0.2 as the subjects of our study. Z3-4.8.1 was released over four years ago in October 2018, while cvc5-0.0.2 was the first official release version of cvc5, released in October 2021. For each release version of the solvers, we feed in the test instances of the 41 fixed bugs reported for Z3 and cvc5 by LAST to check how many bugs also exist in these release versions. If the original formula can trigger the bug in the release version, we consider the bug to have existed in the version. Figure 4 illustrates the fixed bugs that affect the release versions of Z3 and cvc5. Our analysis reveals that while most of the bugs were only present in the trunk versions of the solvers, some bugs had been lurking for a long time. Specifically, three of the bugs in Z3 remained latent for over four years, indicating that these bugs were not exposed by the developers or other SMT fuzzers. Similarly, there were four bugs in cvc5 that remained latent for around two years. In conclusion, our findings demonstrate that LAST is capable of detecting long-latent bugs in SMT solvers, highlighting the practical importance of our technique. In addition, we also submitted a Pull Request to Z3's test suite repository to include the test instances generated by LAST, which was kindly accepted by the developers.[4] It also reflects the significance of the bugs identified by LAST.

C. RQ3: Complementarity Analysis

In this RQ, we aim to investigate whether LAST can complement existing SMT fuzzers from two aspects: (1) the code coverage achieved by the generated formulas, and (2) the bug-exposing capability of the generated formulas.

1) Experimental Setup:

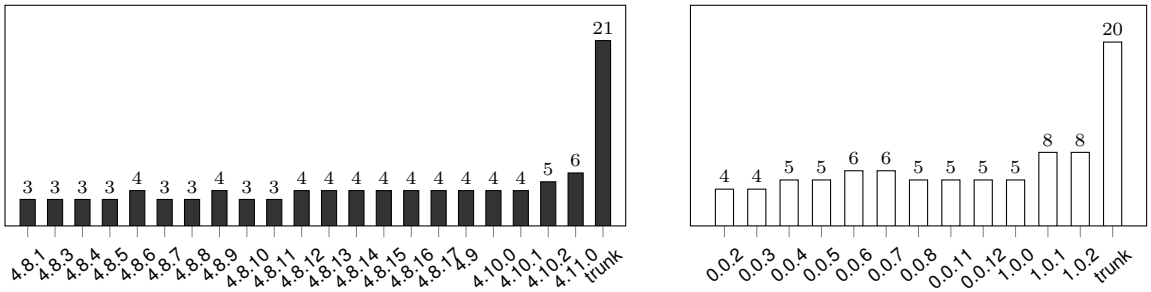
Baselines. We compare LAST with the state-of-the-art SMT fuzzers, namely YinYang [9], STORM [10], OpFuzz [11], TypeFuzz [12], and HistFuzz [13], from the aforementioned aspects. All of these fuzzers are open-source and have been proposed in recent years. For our evaluation, we utilize the latest versions of these fuzzers available in their respective GitHub repositories, employing the default configurations as specified in the original papers.

Code Coverage. To assess the code coverage achieved by each fuzzer, we sample 100 formulas from the SMT-LIB benchmarks at random and utilize the four fuzzers, namely YinYang, STORM, OpFuzz, and TypeFuzz, to generate 10 mutants for each formula, resulting in 1,000 formulas for each fuzzer. Besides, we employ HistFuzz and LAST to generate the same number of formulas (i.e., 1,000), respectively. We then feed the selected seeds with the different groups of formulas generated by the fuzzers to the solvers and utilize the gcov[5] tool to collect the solvers' code coverage. To examine the new code coverage achieved by LAST compared to the baselines, we also analyze the code coverage achieved by each individual baseline fuzzer and LAST simultaneously. This experiment is consistent with the setup of prior studies [12], [13]. Additionally, we repeat this experiment 10 times and calculate the average of code coverage to mitigate the influence of randomness.

Bug Exposing. We re-run each fuzzer for 24 hours and record the number of unique known bugs found by each fuzzer. To identify the unique bugs, we employed the *Correcting Commit* approach [47], [13]. Specifically, we feed a potential bug-triggering instance 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 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 bugs. In practice, we exploit binary search to accelerate the process of identifying

[4] https://github.com/Z3Prover/z3test/pull/48

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



(a) Z3 (b) cvc5

Fig. 4: Fixed bugs that affect the corresponding release versions of Z3 and cvc5.

TABLE IV: Code coverage achieved by baselines, LAST, and their combinations on Z3 and cvc5.

	Z3		cvc5	
	line	*function*	*line*	*function*
LAST	30.4%	27.8%	27.5%	43.6%
OpFuzz	20.8%	19.5%	22.1%	40.0%
+ LAST	30.8%	28.1%	27.8%	44.2%
TypeFuzz	20.7%	19.7%	21.6%	38.6%
+ LAST	30.4%	27.9%	27.5%	43.7%
STORM	23.5%	22.3%	23.3%	41.4%
+ LAST	31.5%	28.8%	28.5%	44.9%
YinYang	21.2%	20.2%	22.0%	38.7%
+ LAST	30.5%	27.9%	27.6%	43.8%
HistFuzz	30.9%	28.1%	28.8%	45.4%
+ LAST	33.2%	29.7%	30.3%	46.5%

the correcting commits. It is important to note that in this experiment, we focus on using fuzzing tools to find known bugs that have already been resolved, rather than finding new bugs, which enables us to conveniently determine the number of unique bugs.

Targeted solvers. In this experiment, we choose Z3 and cvc5 as the subjects for a fair comparison with other fuzzers. We select these two solvers because they have been used as subjects in prior studies. To evaluate the bug-exposing capability of LAST, we specifically select Z3-4.8.5 and CVC4-1.7 (the predecessor of cvc5), following the experimental setup of a prior study [13]. This is because our aim is to find known bugs in this experiment, and the baselines have already found many bugs for these two releases and subsequent versions. Besides, to mitigate the potential influence of data leakage, we also include the latest release version of the solvers (i.e., Z3-4.12.2 and cvc5-1.0.5) in our experiment. The bug-triggering inputs used for training are primarily reported and resolved prior to the release of these versions, thereby avoiding the data leakage problem. Additionally, the code coverage experiment is also conducted on the latest versions of the solvers, and we only adopt the default mode of the solvers in this experiment.

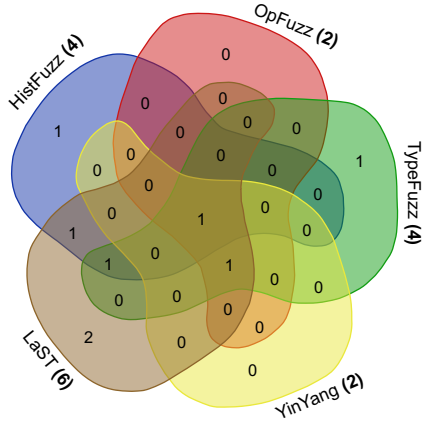


Fig. 5: The distribution of bugs detected by LAST and the baselines on previous versions of solvers.

2) Results:

Table IV displays the code coverage attained by the fuzzers on Z3 and cvc5. Specifically, LAST achieves line coverage of 30.4% and 27.5% on Z3 and cvc5, respectively, surpassing the majority of baselines, such as YinYang, STORM, OpFuzz, and TypeFuzz. Additionally, the combination of LAST and other fuzzers consistently achieves higher code coverage than a single baseline alone, indicating LAST's ability to explore challenging code locations.

Figure 5 shows the bug distribution identified by LAST and the baselines on previously selected solver versions. The results indicate that LAST discovers six bugs within 24 hours, surpassing the baselines in bug detection. Particularly, two of the six bugs are detected only by LAST, highlighting its orthogonal nature concerning existing SMT solver testing techniques. Notably, STORM fails to identify any bugs within the same timeframe, and thus we exclude it from the analysis presented in the figure. For the experiment on the latest versions, as shown in Figure 6, LAST detects five bugs in Z3-4.12.2 and cvc5-1.0.5 within 24 hours. However, HistFuzz only discovers two bugs and YinYang finds one bug, while the other tools fail to find any bugs. These results indicate that LAST outperforms the baselines in terms of bug detection on both previous and latest solver releases.

In conclusion, LAST complements existing SMT solver

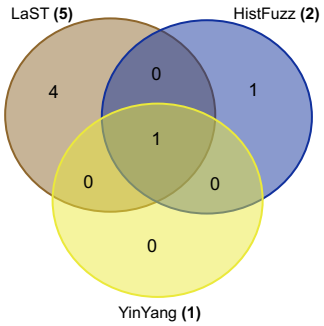


Fig. 6: The distribution of bugs detected by LAST and the baselines on the latest versions of solvers.

testing techniques in terms of both the achieved code coverage and bug-finding capacity.

D. RQ4: Ablation Study

This RQ aims to investigate the contributions of the components to the effectiveness of LAST, including the finetuning process and the data augmentation technique. To this end, we conduct an ablation study using two variants of LAST, denoted as LAST-woFT, which excludes the finetuning process, and LAST-woDF, which excludes both the data augmentation technique and the finetuning process, respectively. We carry out an analysis of the achieved code coverage of LAST-woFT and LAST-woDF. The experimental setup for this analysis is the same as that used for RQ3, which is described in Section IV-C.

Table V presents the results of this experiment. We observe that the coverage achieved by LAST-woFT is lower than that of LAST, which illustrates that the finetuning process can improve the effectiveness of LAST in terms of code coverage. Moreover, the coverage further decreases when we remove the data augmentation technique, indicating its effectiveness. In conclusion, we deduce that both the finetuning process and the data augmentation technique contribute to the effectiveness of LAST in terms of code coverage.

TABLE V: Comparison of LAST and its variants in terms of code coverage.

	Z3		**cvc5**	
	line	*function*	*line*	*function*
LAST	30.4%	27.8%	27.5%	43.6%
LAST-woFT	28.6%	26.3%	26.2%	42.7%
LAST-woDF	26.7%	25.2%	26.1%	41.9%

V. DISCUSSION

A. Case Study

It is noteworthy that a majority of the bugs identified by LAST are related to crashes, which warrants further investigation. To gain a deeper understanding of this trend, we conduct an analysis of the bug types reported by users in Z3 and cvc5's issue trackers. Our analysis reveals that approximately 60% of the bugs in Z3 are crashes, while soundness bugs account for no more than 20% of the total bugs. Similarly, in the case of cvc5, over 70% of the bugs are crashes, with soundness issues comprising less than 10% of the total bugs. Therefore, we can conclude that crashes are the predominant type of bugs in SMT solvers, and it is reasonable to expect that LAST will identify a significant number of such bugs. This expectation is further supported by the fact that LAST is trained using historical bug-triggering formulas, many of which consist of crash bugs. In addition to its proficiency in identifying crash bugs, LAST has demonstrated an ability to generate interesting edge cases that were not covered by existing approaches. For instance, LAST generates a formula related to String theory containing a UTF-8 character that was not included by the existing test suite of Z3. This formula triggers a crash bug in Z3.[6] Although the root cause of this issue is the inadequate support of Z3 for all UTF-8 characters, developers are actively working to enhance the functionality. Therefore, we believe that LAST will continue to be a valuable asset in the future.

B. Threats to Validity

This study is subject to several potential threats to validity that should be considered when interpreting the results. First, it is shown that the formulas generated by LAST are not always well-formed, which could potentially affect the experimental results. However, as noted by the developers of the SMT solvers [8], even invalid inputs can be useful for fuzzing SMT solvers to test their ability to handle unexpected or malformed inputs. Therefore, we view this as a potential threat to internal validity, but one that is largely mitigated by the goals of our study. Second, the choice of target solvers used in our experiments may also introduce a threat to the validity of our results. To address this, we have selected three solvers - Z3, cvc5, and Bitwuzla - that represent a mix of mature and newer solvers. This allows us to evaluate the effectiveness of our approach across a range of different solvers and provides a more comprehensive view of its potential impact. Third, we have implemented LAST based on GPT-2, which is a relatively small LLM compared to the latest LLMs such as LLaMA-2 [48]. However, we have demonstrated that LAST is capable of generating effective formulas for SMT solver testing, and we believe that utilizing larger LLMs could further enhance the effectiveness of LAST. Overall, while there are potential threats to validity inherent in our study, we have made efforts to mitigate them. We believe that our results provide valuable insights into the effectiveness of using pre-trained LLMs for formula generation in SMT solver testing.

C. Future Work

The *retrain-finetune* approach proposed in this study offers a unified framework for leveraging the capabilities of LLMs in software system testing. We believe that this approach can be further expanded to test a wide range of software systems. Specifically, there is potential to extend our unified framework

[6] https://github.com/Z3Prover/z3/issues/6655

to other software systems, especially those with inputs that deviate from conventional programming languages, such as Database Management Systems and Smart Contracts. Additionally, it would be valuable to investigate the effectiveness of alternative LLMs in the context of SMT solver testing, including both open-source and commercial ones.

VI. Related Work

A. SMT Solver Fuzzing

As a cornerstone of formal methods, SMT solvers have been widely used in various domains in software engineering [49], [50], [51], [52]. Therefore, the correctness and robustness of SMT solvers become more significant, and numerous techniques have been proposed to test SMT solvers accordingly. These techniques can be roughly divided into two categories: generation-based and mutation-based. For generation-based techniques, Robert Brummayer and Armin Biere initially proposed a grammar-based blackbox fuzzing tool, FuzzSMT [7], to test SMT solvers. More recently, Niemetz et al. [8] proposed model-based fuzzers, Murxla, to validate SMT solvers by generating sequences of API calls. However, these generation-based techniques are limited by predefined generation strategies, which may not be able to generate diverse test instances. For mutation-based techniques, Winterer et al. [9] use the method of semantic fusion to synthesize two formulas and obtain satisfiability-preserving formulas. STORM [10] obtains satisfiable formulas with Boolean structures different from the original seed through construction. OpFuzz [11] tests SMT solvers by simply mutating operators and achieves satisfactory results. Sparrow [42] leverages the idea of approximations to mutate operators and generate formulas with test oracles. To further broaden the mutation space, Typefuzz [12] improved OpFuzz by combining mutation and generation to yield more diverse mutants. HistFuzz [13] extracts the logical structures of historical bug-triggering formulas and instantiates them with different atomic formulas to generate new formulas. Although these techniques have achieved inspiring results, they also have some limitations since they typically mutate formulas from specific perspectives and are incapable of generating sufficiently diverse mutants. Therefore, we propose a learning-based technique, LAST, to generate a large number of diverse test instances for SMT solver fuzzing. LAST is retrained on formulas in SMT-LIB benchmarks and finetuned on bug-triggering formulas, which is expected to generate effective and diverse test formulas.

B. Deep Learning for Fuzzing

Deep learning-based fuzzing is a novel technique that exploits the power of deep neural networks to generate test inputs for software systems. Fuzzing, a widely used method for detecting bugs and vulnerabilities in software, involves providing random or semi-random inputs and observing the system's behavior. However, traditional fuzzers often struggle to generate massive, diverse, and valid inputs that can adequately exercise the vast codebase of modern software systems. With recent advancements in pre-trained language models, deep learning-based fuzzing has emerged as a promising technique to tackle this challenge. Fuzzing techniques based on LLMs have also shown promising results in several domains. For instance, Ye et al. [17] propose a fuzzing technique that fine-tunes GPT-2 to generate inputs for JavaScript engines. More recently, Deng et al. [28] employ LLMs to fuzz deep learning libraries. They use a generative model (i.e., Codex) to generate programs that invoke deep learning libraries and an infilling model (i.e., InCoder) to further mutate the generated programs. In addition to leveraging LLMs in a zero-shot manner, Deng et al. [29] also propose to finetune LLMs for deep learning library fuzzing and achieve satisfactory effects.

In this study, we apply LLMs to fuzz SMT solvers. To unleash the potential of generating effective SMT instances, we propose a novel *retrain-finetune* pipeline to transfer the power of LLMs to SMT solver fuzzing. Our tool, LAST, can detect authentic bugs in SMT solvers, providing a new perspective for SMT solver fuzzing.

VII. Conclusion

This study presents a novel approach that harnesses large pre-trained language models for generating SMT formulas to fuzz SMT solvers. To fully exploit the capabilities of language models for generating SMT formulas, we employ a *retrain-finetune* pipeline that adapts the language models to the domain of SMT solver. Furthermore, to enhance the effectiveness of the trained model, we utilize data augmentation techniques via two formula mutation strategies, namely, diversity-oriented and semantic-preserving mutations. Our approach is implemented as a practical fuzzing tool named LAST. To evaluate the efficacy of LAST, we conducted a bug-hunting campaign aimed at exposing real bugs in the state-of-the-art SMT solvers, including Z3, cvc5, and Bitwuzla. The results of our campaign reveal that LAST reported a total of 65 bugs for the solvers, of which 45 were subsequently fixed by the developers. Notably, LAST uncovered several critical and long-latent bugs in the SMT solvers. The experimental results demonstrate the potential of leveraging large pre-trained language models for fuzzing SMT solvers.

Acknowledgement

We would like to extend our heartfelt gratitude to all the developers of Z3, cvc5, and Bitwuzla for their valuable information and efforts in addressing the bug reports we submitted. We are also grateful to the anonymous reviewers for their insightful comments and suggestions. Furthermore, we would like to express our appreciation to the authors of YinYang and STORM for generously sharing their artifacts. This work is mainly supported by the National Natural Science Foundation of China (Grant No. 62072194, No. 62172205, No. 62002125), the Natural Science Foundation of Jiangsu Province (No. SBK2023022696), the CCF-Huawei Populus euphratica Fund (Grant No. CCF-HuaweiSY2022007), and the Young Elite Scientists Sponsorship Program by CAST (Grant No. 2021QNRC001).

REFERENCES

[1] C. Barrett and C. Tinelli, "Satisfiability modulo theories," in *Handbook of model checking*. Springer, 2018, pp. 305–343.

[2] G. Soltana, M. Sabetzadeh, and L. C. Briand, "Practical constraint solving for generating system test data," *ACM Trans. Softw. Eng. Methodol.*, vol. 29, no. 2, pp. 11:1–11:48, 2020. [Online]. Available: https://doi.org/10.1145/3381032

[3] E. Kang, S. Lafortune, and S. Tripakis, "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*, ser. Lecture Notes in Computer Science, I. Dillig and S. Tasiran, Eds., vol. 11561. Springer, 2019, pp. 219–237. [Online]. Available: https://doi.org/10.1007/978-3-030-25540-4_12

[4] M. Y. R. Gadelha, E. Steffinlongo, L. C. Cordeiro, B. Fischer, and D. A. Nicole, "Smt-based refutation of spurious bug reports in the clang static analyzer," in *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, J. M. Atlee, T. Bultan, and J. Whittle, Eds. IEEE / ACM, 2019, pp. 11–14. [Online]. Available: https://doi.org/10.1109/ICSE-Companion.2019.00026

[5] B. Cook, "Formal reasoning about the security of amazon web services," in *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*, ser. Lecture Notes in Computer Science, H. Chockler and G. Weissenbacher, Eds., vol. 10981. Springer, 2018, pp. 38–47. [Online]. Available: https://doi.org/10.1007/978-3-319-96145-3_3

[6] P. Yao, H. Huang, W. Tang, Q. Shi, R. Wu, and C. Zhang, "Fuzzing SMT solvers via two-dimensional input space exploration," in *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021*, C. Cadar and X. Zhang, Eds. ACM, 2021, pp. 322–335. [Online]. Available: https://doi.org/10.1145/3460319.3464803

[7] R. Brummayer and A. Biere, "Fuzzing and delta-debugging smt solvers," in *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*, ser. SMT '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 1–5. [Online]. Available: https://doi.org/10.1145/1670412.1670413

[8] A. Niemetz, M. Preiner, and C. W. Barrett, "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*, ser. Lecture Notes in Computer Science, S. Shoham and Y. Vizel, Eds., vol. 13372. Springer, 2022, pp. 92–106. [Online]. Available: https://doi.org/10.1007/978-3-031-13188-2_5

[9] D. Winterer, C. Zhang, and Z. Su, "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*, A. F. Donaldson and E. Torlak, Eds. ACM, 2020, pp. 718–730. [Online]. Available: https://doi.org/10.1145/3385412.3385985

[10] M. N. Mansur, M. Christakis, V. Wüstholz, and F. Zhang, "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*, 2020, pp. 701–712.

[11] D. Winterer, C. Zhang, and Z. Su, "On the unusual effectiveness of type-aware operator mutations for testing SMT solvers," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, pp. 193:1–193:25, 2020. [Online]. Available: https://doi.org/10.1145/3428261

[12] J. Park, D. Winterer, C. Zhang, and Z. Su, "Generative type-aware mutation for testing SMT solvers," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, pp. 1–19, 2021. [Online]. Available: https://doi.org/10.1145/3485529

[13] M. Sun, Y. Yang, M. Wen, Y. Wang, Y. Zhou, and H. Jin, "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, 2023, pp. 69–81. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00018

[14] C. Niu, C. Li, V. Ng, D. Chen, J. Ge, and B. Luo, "An empirical comparison of pre-trained models of source code," *CoRR*, vol. abs/2302.04026, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2302.04026

[15] Z. Zeng, H. Tan, H. Zhang, J. Li, Y. Zhang, and L. Zhang, "An extensive study on pre-trained models for program understanding and generation," in *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*, S. Ryu and Y. Smaragdakis, Eds. ACM, 2022, pp. 39–51. [Online]. Available: https://doi.org/10.1145/3533767.3534390

[16] C. S. Xia, Y. Wei, and L. Zhang, "Practical program repair in the era of large pre-trained language models," *CoRR*, vol. abs/2210.14179, 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2210.14179

[17] G. Ye, Z. Tang, S. H. Tan, S. Huang, D. Fang, X. Sun, L. Bian, H. Wang, and Z. Wang, "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*, S. N. Freund and E. Yahav, Eds. ACM, 2021, pp. 435–450. [Online]. Available: https://doi.org/10.1145/3453483.3454054

[18] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. T. Devanbu, "On the naturalness of software," in *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, M. Glinz, G. C. Murphy, and M. Pezzè, Eds. IEEE Computer Society, 2012, pp. 837–847. [Online]. Available: https://doi.org/10.1109/ICSE.2012.6227135

[19] D. Wang, Z. Jia, S. Li, Y. Yu, Y. Xiong, W. Dong, and X. Liao, "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, 2022, pp. 287–298. [Online]. Available: https://doi.org/10.1145/3510003.3510062

[20] SMT-LIB. (2021) SMT-LIB Benchmarks. [Online]. Available: http://smtlib.cs.uiowa.edu/benchmarks.shtml

[21] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[22] OpenAI. (2019) Gpt-2: 1.5b release. [Online]. Available: https://openai.com/blog/gpt-2-1-5b-release/

[23] H. Barbosa, C. W. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar, "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*, ser. Lecture Notes in Computer Science, D. Fisman and G. Rosu, Eds., vol. 13243. Springer, 2022, pp. 415–442. [Online]. Available: https://doi.org/10.1007/978-3-030-99524-9_24

[24] A. Niemetz and M. Preiner, "Bitwuzla," in *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part II*, ser. Lecture Notes in Computer Science, C. Enea and A. Lal, Eds., vol. 13965. Springer, 2023, pp. 3–17. [Online]. Available: https://doi.org/10.1007/978-3-031-37703-7_1

[25] S. Ranise and C. Tinelli, "The smt-lib format: An initial proposal," in *Proceedings of the 1st Workshop on Pragmatics of Decision Procedures in Automated Reasoning*. Citeseer, 2003.

[26] C. Barrett, P. Fontaine, and C. Tinelli. (2022) The satisfiability modulo theories library (SMT-LIB). [Online]. Available: http://smtlib.cs.uiowa.edu/

[27] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett, Eds., 2017, pp. 5998–6008. [Online]. Available: https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

[28] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, "Fuzzing deep-learning libraries via large language models," *CoRR*, vol. abs/2212.14834, 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2212.14834

[29] Y. Deng, C. S. Xia, C. Yang, S. D. Zhang, S. Yang, and L. Zhang, "Large language models are edge-case fuzzers: Testing deep learning libraries via fuzzgpt," *CoRR*, vol. abs/2304.02014, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2304.02014

[30] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*, P. L. Bartlett, F. C. N. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds., 2012, pp. 1106–1114. [Online]. Available: https://proceedings.neurips.cc/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html

[31] S. Yang, W. Xiao, M. Zhang, S. Guo, J. Zhao, and F. Shen, "Image data augmentation for deep learning: A survey," *CoRR*, vol. abs/2204.08610, 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2204.08610

[32] N. Yefet, U. Alon, and E. Yahav, "Adversarial examples for models of code," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, pp. 162:1–162:30, 2020. [Online]. Available: https://doi.org/10.1145/3428230

[33] E. Quiring, A. Maier, and K. Rieck, "Misleading authorship attribution of source code using adversarial learning," in *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, N. Heninger and P. Traynor, Eds. USENIX Association, 2019, pp. 479–496. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/quiring

[34] HistFuzz. (2023). [Online]. Available: https://github.com/CGCL-codes/HistFuzz

[35] Y. Zhao, Z. Wang, J. Chen, M. Liu, M. Wu, Y. Zhang, and L. Zhang, "History-driven test program synthesis for jvm testing," in *44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 21-29, 2022*. IEEE, 2022. [Online]. Available: https://doi.org/10.1145/3510003.3510059

[36] S. Lee, H. Han, S. K. Cha, and S. Son, "Montage: A neural network language model-guided javascript engine fuzzer," in *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, S. Capkun and F. Roesner, Eds. USENIX Association, 2020, pp. 2613–2630. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/lee-suyoung

[37] H. Zhong, "Enriching compiler testing with real program from bug report," in *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 2022, pp. 40:1–40:12. [Online]. Available: https://doi.org/10.1145/3551349.3556894

[38] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2019.

[39] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*. The Association for Computer Linguistics, 2016. [Online]. Available: https://doi.org/10.18653/v1/p16-1162

[40] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015. [Online]. Available: http://arxiv.org/abs/1412.6980

[41] M. Deng, J. Wang, C. Hsieh, Y. Wang, H. Guo, T. Shu, M. Song, E. P. Xing, and Z. Hu, "Rlprompt: Optimizing discrete text prompts with reinforcement learning," in *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7-11, 2022*, Y. Goldberg, Z. Kozareva, and Y. Zhang, Eds. Association for Computational Linguistics, 2022, pp. 3369–3391. [Online]. Available: https://aclanthology.org/2022.emnlp-main.222

[42] P. Yao, H. Huang, W. Tang, Q. Shi, R. Wu, and C. Zhang, "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*, D. Spinellis, G. Gousios, M. Chechik, and M. D. Penta, Eds. ACM, 2021, pp. 1141–1153. [Online]. Available: https://doi.org/10.1145/3468264.3468540

[43] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker," in *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, G. Heiser and W. C. Hsieh, Eds. USENIX Association, 2012, pp. 309–318. [Online]. Available: https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[44] cvc5. (2022) cvc5 fuzzing guidelines. [Online]. Available: https://github.com/cvc5/cvc5/wiki/Fuzzing-cvc5

[45] G. Kremer, A. Niemetz, and M. Preiner, "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*, ser. Lecture Notes in Computer Science, A. Silva and K. R. M. Leino, Eds., vol. 12760. Springer, 2021, pp. 231–242. [Online]. Available: https://doi.org/10.1007/978-3-030-81688-9_11

[46] "pydelta: delta debugging for smt-lib," https://github.com/nafur/pydelta.

[47] J. Chen, W. Hu, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie, "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*, L. K. Dillon, W. Visser, and L. A. Williams, Eds. ACM, 2016, pp. 180–190. [Online]. Available: https://doi.org/10.1145/2884781.2884878

[48] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. Bikel, L. Blecher, C. Canton-Ferrer, M. Chen, G. Cucurull, D. Esiobu, J. Fernandes, J. Fu, W. Fu, B. Fuller, C. Gao, V. Goswami, N. Goyal, A. Hartshorn, S. Hosseini, R. Hou, H. Inan, M. Kardas, V. Kerkez, M. Khabsa, I. Kloumann, A. Korenev, P. S. Koura, M. Lachaux, T. Lavril, J. Lee, D. Liskovich, Y. Lu, Y. Mao, X. Martinet, T. Mihaylov, P. Mishra, I. Molybog, Y. Nie, A. Poulton, J. Reizenstein, R. Rungta, K. Saladi, A. Schelten, R. Silva, E. M. Smith, R. Subramanian, X. E. Tan, B. Tang, R. Taylor, A. Williams, J. X. Kuan, P. Xu, Z. Yan, I. Zarov, Y. Zhang, A. Fan, M. Kambadur, S. Narang, A. Rodriguez, R. Stojnic, S. Edunov, and T. Scialom, "Llama 2: Open foundation and fine-tuned chat models," *CoRR*, vol. abs/2307.09288, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2307.09288

[49] P. Tolmach, Y. Li, S. Lin, Y. Liu, and Z. Li, "A survey of smart contract formal specification and verification," *ACM Comput. Surv.*, vol. 54, no. 7, pp. 148:1–148:38, 2022. [Online]. Available: https://doi.org/10.1145/3464421

[50] R. Tzoref and O. Grumberg, "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*, ser. Lecture Notes in Computer Science, T. Ball and R. B. Jones, Eds., vol. 4144. Springer, 2006, pp. 190–204. [Online]. Available: https://doi.org/10.1007/11817963_20

[51] R. Degiovanni, D. Alrajeh, N. Aguirre, and S. Uchitel, "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*, P. Jalote, L. C. Briand, and A. van der Hoek, Eds. ACM, 2014, pp. 129–139. [Online]. Available: https://doi.org/10.1145/2568225.2568323

[52] M. Mendonça, A. Wasowski, and K. Czarnecki, "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*, ser. ACM International Conference Proceeding Series, D. Muthig and J. D. McGregor, Eds., vol. 446. ACM, 2009, pp. 231–240. [Online]. Available: https://dl.acm.org/citation.cfm?id=1753267