Towards Understanding the Effectiveness of Large Language Models on Directed Test Input Generation

Zongze Jiang[*][†]
Huazhong University of Science and Technology
Wuhan, China
jiangzongze@hust.edu.cn

Ming Wen[*][†][‡]
Huazhong University of Science and Technology
Wuhan, China
mwenaa@hust.edu.cn

Jialun Cao
The Hong Kong University of Science and Technology
Hong Kong, China
jcaoap@cse.ust.hk

Xuanhua Shi[*][§]
Huazhong University of Science and Technology
Wuhan, China
xhshi@hust.edu.cn

Hai Jin[*][§]
Huazhong University of Science and Technology
Wuhan, China
hjin@hust.edu.cn

Abstract

Automatic testing has garnered significant attention and success over the past few decades. Techniques such as unit testing and coverage-guided fuzzing have revealed numerous critical software bugs and vulnerabilities. However, a long-standing, formidable challenge for existing techniques is how to achieve higher testing coverage. Constraint-based techniques, such as symbolic execution and concolic testing, have been well-explored and integrated into the existing approaches. With the popularity of Large Language Models (LLMs), recent research efforts to design tailored prompts to generate inputs that can reach more uncovered target branches. However, the effectiveness of using LLMs for generating such directed inputs and the comparison with the proven constraint-based solutions has not been systematically explored.

To bridge this gap, we conduct the first systematic study on the mainstream LLMs and constraint-based tools for directed input generation with a comparative perspective. We find that LLMs such as ChatGPT are comparable to or even better than the constraint-based tools, succeeding in 43.40%-58.57% samples in our dataset. Meanwhile, there are also limitations for LLMs in specific scenarios such as sequential calculation, where constraint-based tools are in a position of strength. Based on these findings, we propose a simple yet effective method to combine these two types of tools and implement a prototype based on *ChatGPT* and constraint-based tools. Our evaluation shows that our approach can outperform the baselines by 1.4x to 2.3x relatively. We believe our study can provide novel insights into directed input generation using LLMs, and our findings are essential for future testing research.

Keywords

LLM, Symbolic Execution, Directed Input Generation

[*]National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Huazhong University of Science and Technology (HUST), Wuhan, 430074, China
[†]Hubei Engineering Research Center on Big Data Security, Hubei Key Laboratory of Distributed System Security, School of Cyber Science and Engineering, HUST, Wuhan, 430074, China
[‡]Corresponding author. Ming Wen is also affiliated with Wuhan JinYinHu Laboratory.
[§]Cluster and Grid Computing Lab, School of Computer Science and Technology, HUST, Wuhan, 430074, China

1 Introduction

Automated testing is essential to guarantee the reliability and security of software systems and has gained significant attention and success over the past decades. Substantial automated testing techniques such as fuzzing [8, 21], property-guided random testing [15, 34], symbolic execution [6, 60], and search-based evolutionary [3, 22, 23] techniques have been proposed and extensively studied. These techniques all face a long-standing formidable challenge, i.e., ***how to achieve higher testing coverage***. Recent studies have proposed hybrid methods to integrate the respective advantages of different approaches [6, 33, 65]. For example, to address the challenge of struggling to cover certain complex constrained branches for random approaches in grey-box fuzzing, symbolic and concolic (*a.k.a.* dynamic symbolic) execution techniques have been incorporated as a complementary component [37, 65].

These constraint-based tools (i.e., symbolic or concolic execution tools) play a crucial role in the hybrid fuzzing approaches for their special ability to precisely explore and generate inputs for uncovered code or paths, which is known as directed input generation (*a.k.a.* targeted input generation) [19, 39]. These techniques solve the constraints collected on the target execution path, thereby obtaining the specific inputs that can reach the targets. Despite their effectiveness, the practical usefulness is still compromised in some aspects [6, 60]. For example, ***path explosion*** occurs with the increase of code complexity, i.e., every conditional statement can potentially double the number of paths to explore. Also, these tools struggle with code that interacts with ***external systems or channels***, such as operating system calls and network communications. For example, as shown in Figure 1, the input `s` is passed to the `system` function, which will recognize the input string as a shell command and execute it. Any valid Linux command, e.g., `'ls'`, could return `0` and cover the target line (L_4) (See Section 2). However, generating such input is challenging for traditional tools as the constraints in `system` are unrealistic to be modeled.

Recently, the huge success of Large Language Models (LLMs) has changed the paradigm of many software engineering tasks such as program repair [20, 29, 41, 59], code generation [11, 48], vulnerability detection [28, 45, 53] and *test generation* [18, 30, 46, 49, 56, 58].

```
    int execute(char* s) {
        int ret = system(s);
        if(ret == 0){
            // target line
        }
    }
```

Figure 1: An example that symbolic execution tools cannot solve while LLMs can.

```
1   int arr(char *s){
2       int symvar = s[0] - 48;
3       int ary[] ={1,2,3,4,5};
4       if (ary[symvar%5] == 5){
5           // target line
6       }
7   }
```

Figure 2: An example that LLMs cannot solve while symbolic execution tools can.

Previous studies have shown that LLMs are highly competitive compared to traditional search-based tools and even outperform them in certain aspects (e.g., readability) on test case generation [52]. However, LLMs also suffer from similar coverage limitation problems, as pointed out by recent studies [2, 43, 47, 56]. To improve this, some works naturally turn to ***directed input generation by LLMs*** and have gained better performance. Specifically, more contexts are fed into LLMs to direct test generation to the uncovered branches or statements, including ***the context in natural language*** (*documentation* and *error messages* are provided to direct LLMs to cover certain *properties* or *circumstances* [44, 55]), and ***the context in code*** [16, 43, 47, 49, 62] (*unreached branches* or *executable paths* are provided to guide input generation for higher coverage).

Yet, are LLMs a panacea for such directed test input generation? Although LLMs have impressive generation capabilities, it is easy for them to generate correct input in the example in Figure 1, Figure 2 showcases an example where LLMs fail to generate correct inputs that can reach the target line. Though the problem appears simple, to generate the directed input to cover the target (L_5), LLMs need to reason about the modulo (%), array index operations, and arithmetic calculation. Neither *ChatGPT* nor *CodeLlama* can generate the directed input. Traditional tools, on the contrary, can solve the constraints easily and generate the correct input (i.e., "4").

Figure 1 and Figure 2 are actually simple examples while ***the stark different performance of LLM and traditional tools*** (See Section 2) motivates us to further systematically explore the performance of LLMs on directed test input generation tasks. More importantly, we aim to compare their effectiveness with traditional constraint-based tools to understand their respective strengths and weaknesses and investigate their potential to complement each other for better performance. To achieve these goals, we first select the existing symbolic execution benchmark *logic_bombs* [60] as our dataset. It contains carefully manually curated samples of targeted input generation challenges categorized by code scenarios. It is utilized to explore the limitations of symbolic/concolic techniques in previous works [60, 61]. Furthermore, to enable more comprehensive evaluations, we transform the widely used dataset for large code models, *HumanEval* [14], and its multilingual version, *HumanEval-X* [64] to construct a new dataset *PathEval*, which contains over 2000 samples in total. Based on these two datasets, we conduct extensive experiments on both *ChatGPT* and open-source LLMs (*StarCoder2*, *CodeLlama*, *CodeQwen*), together with constraint-based tools (*Angr*, *KLEE*, *CrossHair*, *EvoSuite*) in terms of four mainstream programming languages (C, C++, Java, Python).

Evaluation results reveal a surprising finding that ***ChatGPT*** **compares favorably or even superior to constraint-based tools** for directed input generation. Specifically, *ChatGPT* passes 51.21%-58.57% samples in three subsets of different languages on *PathEval*, only 0.94% less than *KLEE* [9] on C++ but significantly higher than other tools. Futhermore, on *logic_bombs*, *ChatGPT* outperforms both *Angr* and *KLEE* by 15.10% and 18.87%. However, through more in-depth research, we find that directly concluding that LLMs are superior to constraint-based tools, or the other way around, is inappropriate. ***The relative strengths and weaknesses of these approaches are more nuanced and context-dependent.*** We further perform statistical analysis and case studies on the categorized samples from aspects of code scenarios, input/output features and generalizability. Our findings support our argument. For instance, in handling code that requires external knowledge, such as parallel program communication, *ChatGPT* demonstrates exceptional dominance, while traditional tools perform better in scenarios demanding more precise calculations (e.g., specific length strings, array index). Furthermore, we discover that a considerable portion of the samples passed by *ChatGPT* and *KLEE* are unique, providing strong evidence for our proposition.

Driven by the above findings, we further explore ***the feasibility of incorporating LLMs and traditional tools*** for better directed test input generation. At a high level, we query LLMs to generate directed inputs and summarize the characteristics of the inputs (e.g., input types and structure), and further utilize this information as feedback and generate more flexible harnesses (compared to fixed templates) to guide traditional tools better. We implement our idea on *ChatGPT* and *KLEE/CrossHair* and conduct experiments on *PathEval*. On the C++ dataset, our approach passes 85.83% samples, outperforming *KLEE* by 44.22%, and *GPT-3.5* by 46.54% relatively. It is notable that there are 28 new successes that both failed for *ChatGPT* and *KLEE* in previous experiments. On the Python dataset, the overall pass rate is 74.43%, outperforming the baselines by 39.00% and 128.81% relatively and obtaining 74 unique successes.

In summary, this study makes the following major contributions:

- ***Originality:*** To the best of our knowledge, we are the first to undertake a systematic study of LLMs for directed test input generation and the comparative analysis of LLMs with constraint-based tools in this particular task.
- ***Findings:*** Our investigations have revealed that *ChatGPT* showcases exceptional directed input generation capabilities, surpassing or catching up with other constraint-based baselines. More importantly, LLMs and constraint-based tools exhibit distinct strengths and weaknesses, and there is potential for integration for better performance.
- ***Perspective:*** Building upon our findings, we propose a straightforward yet effective approach that combines LLMs with

constraint-based tools to prove our point. Evaluation results show that our approach outperforms the baselines.

- ***Open-Source:*** We have open-sourced our artifacts and tool at: **https://github.com/CGCL-codes/PathEval**. We believe our findings, benchmarks, and tools can benefit future research toward LLMs on directed test input generation.

2 Background and Motivation

The goal of ***directed test input*** (*a.k.a.* targeted test input) generation is to automatically generate test inputs with a given target [19, 39]. Targeted inputs are widely used in software engineering tasks such as *bug reproducation* (target is the bug location) [4, 10], *test suite argument* (target is the code to cover) [40, 51] and *combing with other testing tools* to get better overall performance [19, 33].

Constraint-based Approaches. A classical approach to generate such targeted inputs is to leverage ***constraint-based tools*** that utilize *Symbolic Execution* [39] or *Concolic Execution* [19] techniques. Symbolic execution techniques analyze programs by treating inputs as symbolic values rather than concrete data. They then systematically explore different execution paths by solving constraints generated from program conditions. Take the code in Figure 2 as an example. Symbolic Execution tools analyze the function from the entry. Instead of executing with a concrete value of `s`, they define a symbolic value `s1` instead and maintain the constraints at the current position (e.g., $\emptyset$ at entry). The symbolization propagates as the analysis continues (e.g., `symvar` at L_2 will be assign with `s1[0]-48`). When encountering conditional branches, the symbolic execution tool forks the current constraints, and upon entering a particular branch, it updates the constraints according to the corresponding branch condition (e.g., `ary[(s1[0]-48)%5]==5` added at L_5). Then, with a constraint solver (e.g., *Z3*), they can generate a directed input from the entry to the target line. For *Backward Symbolic Execution* [19, 39] tools, this process reverses, starting with the target line and stop and the entry point. *Concolic Execution* (*a.k.a.* dynamic symbolic execution) combines concrete execution with symbolic analysis. Instead of systematic exploration, dynamic symbolic execution collects the constraint during concrete execution to mitigate limitations such as external interactions, the overhead of systematic exploration, etc. Previous work combined grey-box fuzzing with these symbolic execution techniques, delegating hard-to-cover branches that fuzzing struggles with to these specialized tools to achieve better performance [33, 65].

LLM-based Approaches. Since the explosive popularity of *ChatGPT*, LLMs have been increasingly applied across a wide range of downstream tasks in the area of software engineering [27]. Recent works show that LLMs perform promising on many software engineering tasks, as well as test generation [16, 43, 44, 47, 49, 55, 56]. Specific to directed input generation, to improve overall coverage and mutation score metrics, recent studies leverage targeted prompts to drive LLMs to generate inputs to reach hard-to-cover branches or statements, for example, including the target line numbers [43] or constraints along the target execution path [47, 62]. However, due to the hallucination problem, LLMs may confidently generate the wrong inputs [62]. Also, their effectiveness compared with constraint-based tools has not yet been explored.

Motivation Example. Here, we describe the examples mentioned in Section 1 in more detail. For constraint-based tools, we have taken Figure 2 as an example above to illustrate how they can precisely solve constraints and reach the specified targets. However, these tools also face severe limitations in practical usages, as exemplified in Figure 1. In this scenario, reaching the target (i.e., L_4) requires the system return value to be zero. An intuitive solution is to input a valid Linux command, which the system will then execute. If the execution is complete without errors, the system will return 0, thereby reaching the target. However, symbolic/concrete execution cannot generate inputs for such code, as the system function breaks the constraint propagation between the input and output. They cannot extract any useful constraints during the symbolic execution or concolic execution process to solve this problem.

Meanwhile, *ChatGPT*, trained on an extensive corpus including technical documentation and online programming resources, has endowed it with a powerful understanding of the intent of code and the functionality of related knowledge. This allows the model to draw upon this learned information to construct plausible command-line inputs that can satisfy the necessary constraints at a high level. However, we found that *ChatGPT* can hardly generate correct input for the sample in Figure 2, which can be easily generated by symbolic and concolic execution tools. After several attempts, we find that *ChatGPT* is prone to making errors in calculations of array index values, modulo operations, and subtraction operations, which prevent it from success.

The above interesting examples motivate us to investigate the performance of LLMs in targeted input generation tasks and more importantly, compare them against constraint-based techniques, analyze the distinct advantages and limitations of each approach, and finally explore the potential for synergistic combinations.

3 Experimental Setup

3.1 Research Questions

In this study, we aim to understand the effectiveness of large language models on directed test input generation and more importantly, systematically compare their performance with traditional constraint-based tools. To achieve such a goal, we design the following three research questions.

RQ1: How do LLMs perform on directed input generation?

In this RQ, we aim to comprehensively understand the performance of open-source and commercial LLMs on the task of directed input generation. Specifically, we conduct evaluations on two benchmarks containing over 2,000 samples (see Section 3.2). The experiments involve Python, Java, and C++ programming languages under various settings (e.g., number of attempts).

RQ2: Can LLMs outperform traditional constraint-based tools?

In this RQ, we aim to systematically compare LLMs with traditional constraint-based tools to understand their respective strengths and weaknesses. Specifically, we select well-known constraint-based tools supporting multiple programming languages and conduct a comparative analysis with LLMs from various perspectives, including different code scenarios (e.g., external functions, integer/floating point calculations), different input and output types (e.g., string types and aggregate types), unique successful samples, and their performance in worst-case and best-case scenarios.

RQ3: Can we integrate LLMs with traditional tools for better directed test input generation?

Finally, we investigate the feasibility of incorporating LLMs with traditional constraint-based tools to improve the effectiveness in the directed test input generation task. Specifically, we design an approach that combines LLMs and constraint-based tools, implement the corresponding prototype, and evaluate it on the benchmarks to find out whether the performance can be enhanced.

3.2 Benchmark Construction

3.2.1 Existing Benchmark. There are a few existing datasets for symbolic/concolic execution tools, such as *logic_bombs* [60] and *CGC*.[1] The *logic_bombs* dataset encompasses diverse samples across various code scenarios that can impact symbolic execution. However, the primary limitation is its relatively small size. There are only 53 samples in *logic_bombs*, which is insufficient individually to adequately support our comprehensive study. Other available datasets, such as *CGC*, primarily focus on vulnerability-exploiting scenarios, particularly related to memory corruption in the memory-unsafe language (e.g., C/C++), which lacks diversity. Additionally, many samples within these datasets are intentionally constructed, introducing complex control and data flows that are not representative of typical development scenarios. Consequently, it may not accurately reflect real-world conditions. *CruxEval* [25] is a benchmark for LLM code reasoning, understanding and execution. Samples in *CruxEval* are constructed in the form of inferring input from the given output of a function and vice versa. However, this benchmark is generated by *CodeLlama-34b* and small (maximum 13 lines, no complex arithmetic) [25], which is unfit for our task of directed input generation, which aims to reach hard-to-cover code in practical usages. Additionally, constructing a dataset from scratch is also challenging. The main difficulty lies in determining which paths or locations should be targeted, as manual selection or selection using some heuristics can inevitably introduce difficult-to-measure bias.

3.2.2 Benchmark Construction from Existing Dataset. Inspired by the samples in *logic_bombs* and *CruxEval*, we propose a method that transforms existing well-established datasets into directed input generation tasks, as illustrated in Figure 3.

Specifically, we select *HumanEval* [14] as the original dataset. *HumanEval*, introduced by *OpenAI*, is currently the most well-known benchmark for evaluating the performance of LLMs on code tasks [14, 64]. It comprises 164 hand-crafted programming problems and their corresponding unit tests, covering various areas, including language understanding, reasoning, algorithms, mathematics, etc. To gain a multilingual perspective, we also utilize its multilingual counterparts, *HumanEval-X* [64].

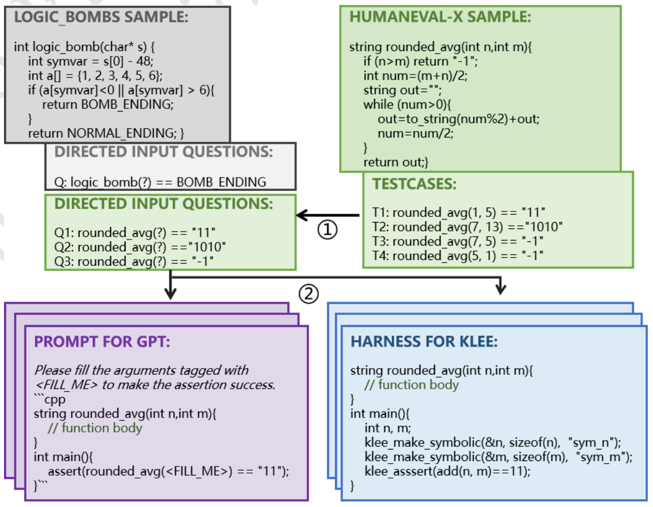
We utilize the test cases in every sample in *HumanEval-X* to construct directed input generation questions. First, for each test case in every sample, we extract the oracles from the corresponding test suites and discard the inputs to construct an inverse-solving problem similar to those samples in *logic_bombs* and *CruxEval* datasets. Then, we mask the input arguments with special tokens to generate the corresponding prompts for LLMs or symbolize them to construct a harness for constraint-based tools to solve it.

[1]https://www.ll.mit.edu/r-d/datasets/cyber-grand-challenge-datasets

Table 1: The datasets used in this study. "Lang." and "Num." refer to the language and number of the samples in the dataset, respectively.

Dataset	Desc.	Lang.	Num.
logic_bombs	Dataset for symbolic execution tools.	C	53
PathEval	Reconstructed from *HumanEval-X*.	C++	741
		Python	747
		Java	744

By transforming existing well-established datasets, our approach avoids the bias introduced in the construction as much as possible. Eventually, from *HumanEval-X*, we have extracted over 2000 samples for 3 languages (i.e. C++, Python, and Java) to serve as our benchmark. To distinguish it from the original dataset, we refer to the transformed dataset as *PathEval*. Such targeted input references contain complex and numerous constraints. Take `T1` in Figure 3 as an example. It includes both obvious constraints (e.g., `n > m` and `(n+m)/2>0`) and more subtle constraints imported by the given output `"11"` (e.g., the `while` should loop exactly twice), which introduces more complex constraints on `n` and `m`.



```
int logic_bomb(char* s) {
    int symvar = s[0] - 48;
    int a[] = {1, 2, 3, 4, 5, 6};
    if (a[symvar]<0 || a[symvar] > 6){
        return BOMB_ENDING;
    }
    return NORMAL_ENDING; }
```

```
string rounded_avg(int n,int m){
    if (n>m) return "-1";
    int num=(m+n)/2;
    string out="";
    while (num>0){
        out=to_string(num%2)+out;
        num=num/2;
    }
    return out;}
```

```
Q: logic_bomb(?) == BOMB_ENDING
```

```
Q1: rounded_avg(?) == "11"
Q2: rounded_avg(?) =="1010"
Q3: rounded_avg(?) == "-1"
```

```
T1: rounded_avg(1, 5) == "11"
T2: rounded_avg(7, 13) =="1010"
T3: rounded_avg(7, 5) == "-1"
T4: rounded_avg(5, 1) == "-1"
```

~~~
Please fill the arguments tagged with
<FILL_ME> to make the assertion success.
```cpp
string rounded_avg(int n,int m){
    // function body
}
int main(){
    assert(rounded_avg(<FILL_ME>) == "11");
}```
~~~

```
string rounded_avg(int n,int m){
    // function body
}
int main(){
    int n, m;
    klee_make_symbolic(&n, sizeof(n),  "sym_n");
    klee_make_symbolic(&m, sizeof(m),  "sym_m");
    klee_asssert(add(n, m)==11);
}
```

Figure 3: Insight and overview of dataset construction from *HumanEval-X*.

Concurrently, we also collect the samples in *logic_bombs* dataset and expand it to accommodate the use of LLMs. While it is relatively small, as we mentioned earlier, the samples labeled with code scenarios can provide valuable insights for our analysis. The composition of our final benchmark is shown in Table 1.

3.2.3 Evaluation Metric. We use the ***pass rate*** as the metric in this study. Briefly, the pass rate represents how many samples in the dataset the tools can pass (i.e., generate an input that can return the targeted output) within M attempts. Let N denote the total number of samples in the dataset. For each sample s_i (where $i = 1, 2, ..., N$), including function f_i and the targeted output o_i, we attempt M times to generate input t_{ij} (where $j = 1, 2, ..., M$). We consider any success of t_{ij} (i.e., $f_i(t_{ij}) = o_i$) as the pass of s_i, denote as p_i:

$$p_i = 1, \bigvee_{j=1}^{M} f_i(t_{ij}) = o_i; and\ 0\ othersie; \quad (1)$$

Then, the pass rate of the overall dataset can be defined as follows:

$$pass\_rate = \sum_{i=1}^{N} p_i / N \quad (2)$$

3.3 Studied LLMs and constraint-base tools

3.3.1 LLMs. For LLMs, we select the top state-of-the-art open-source LLMs from the leaderboard for code-related tasks.[2] Our selection process takes into account factors of computational resource limitations, community prominence, and prior usage in academic works [13, 25, 31, 64]. Additionally, we included *ChatGPT* as a representative of closed-source commercial LLMs.

GPT-3.5 (ChatGPT): As the most well-known and widely used large language chat model, *ChatGPT* has been extensively utilized in previous research studies, with its performance validated by numerous works [1, 12, 17, 26, 52, 57, 63]. Additionally, it offers a lower cost API than *GPT-4* (approximately 1/20), making it more suitable for development as an automated tool. Specifically, we use OpenAI API with the default setting and instruction fashion to prompt *ChatGPT* (See Figure 3 for an example prompt).

StarCoder2 [32, 35]: *StarCoder2* is a family of code generation models with parameter sizes ranging from 3B to 15B. In our experiments, we employ the fill-in-the-middle approach for query. Note that we utilized the 7b model instead of the 15b model, as there is evidence[3] suggesting potential flaws in the performance of the 15b model in the fill-in-the-middle mode, making it less effective compared to the 7B models [35].

CodeLlama [54]: *CodeLlama*, leveraging the advancements of *Llama 2*, showcases exceptional performance, offering cutting-edge capabilities such as superior infilling, the ability to handle large input contexts, and impressive zero-shot instruction following for programming tasks. We select the 7B model with the filling-in-the-middle prompt in our study due to the resource limitations as well as to be consistent with *StarCoder2*.

CodeQwen [5]: *CodeQwen 1.5* is built upon *Qwen 1.5* and was trained on 3 trillion tokens of code data. Within the temporal scope of this work's production, *CodeQwen-1.5* ranks as the top-performing base model on the leaderboard. We utilize the *CodeQwen-1.5-7b-chat* model with the same prompt approach as *GPT-3.5*.

3.3.2 Constraint-based Tools. We chose widely used symbolic execution/dynamic symbolic execution tools that support the Binary, C++, Python, and Java languages, respectively, in this study.

KLEE [9]: *KLEE* is an open-source symbolic execution engine for automated analysis and testing of C and C++ programs. It utilizes static analysis and constraint solving along symbolic execution paths to explore various paths and detect errors and vulnerabilities.

Angr [50]: *Angr* is a powerful binary analysis framework that supports multiple architectures and facilitates tasks like symbolic execution and program analysis. It is widely recognized and popular in CTF competitions, binary analysis, and reverse engineering.

Table 2: The performance (i.e., pass rate) of studied LLMs on the two datasets. The "Lang." column indicates the language of the samples. The "Att." column indicates the results under single and five attempts, respectively. The red coloring columns indicate the gap between the tool's performance and that of *GPT-3.5*.

Dataset	Lang.	Att.	*GPT-3.5 (%)*	*CodeLlama (%)*		*StarCoder2 (%)*		*CodeQwen (%)*	
logic-bomb	C	1	35.85	13.21	-22.64	9.43	-26.42	28.30	-7.55
		5	43.40	15.09	-28.31	13.21	-30.19	37.74	-5.66
PathEval	C++	1	38.06	30.23	-7.83	12.55	-25.51	18.89	-19.16
		5	58.57	36.30	-22.27	22.94	-35.63	28.34	-30.23
	Python	1	36.81	28.66	-8.15	14.46	-22.36	26.51	-10.31
		5	53.55	34.15	-19.40	19.28	-34.27	38.15	-15.39
	Java	1	31.05	11.91	-19.14	10.89	-20.16	23.92	-7.12
		5	51.21	16.38	-34.83	15.19	-36.02	37.10	-14.11

CrossHair[4]: *CrossHair* is a popular (over 900 stars on *Github*) program analysis tool for Python. *CrossHair* utilizes dynamic symbolic execution techniques to explore different execution paths and generate corresponding test cases for users.

EvoSuite [22, 23]: *EvoSuite* is a Java-based tool that automatically generates test cases with assertions for software classes. The latest version of *EvoSuite* also supports generating tests in dynamic symbolic execution (*a.k.a.* concolic execution) mode.

3.3.3 Settings for Different Tools. Here, we provide the detailed settings we utilized for the selected tools during the experiment.

LLMs hyperparameters: We set up *GPT-3.5* with the default configuration of *OpenAI* API. For open-source LLMs, we follow the settings in the guidance[5] and paper [54] in *CodeLlama* (temperature=0.1, top_p= 0.95, max_new_token=100).

LLMs prompt construction: We use the same prompt template shown in Figure 3 to instruct chat models *GPT-3.5* and *CodeQwen*. And for code models, *CodeLlama* and *StarCoder2*, we use their sentinel tokens (`<FILL_ME>` for *CodeLlama*, `<fim_prefix>`, `<fim_suffix>` and `<fim_middle>` for *StarCoder2*) to conduct the fill-in-the-middle prompt [35, 54].

Constraint-based tools setting: We follow the settings in the benchmark of *logic_bombs* [60] to set up *KLEE* and *Angr*. For *CrossHair* and *EvoSuite*, which are unused in *logic_bombs*, we utilize the default settings for dynamic symbolic execution (`crosshair cover` command for *CrossHair* and `-generateSuiteUsingDSE` flag for *EvoSuite*). The timeout is set to 300 seconds for all studied tools.

We have performed the experiment of the open-source LLMs with a single NVIDIA Tesla A100-PCIe with 40GB Graphics RAM and evaluated the selected constraint-based tool on a server with a 96-core Intel Xeon Gold 6248R CPU and 256GB RAM.

4 Evaluation Result

4.1 RQ1: Effectiveness of LLM

In this RQ, we aim to investigate the effectiveness of LLMs in generating directed inputs. We evaluate the selected LLMs on our multilingual dataset. For every sample, we utilize LLMs to generate

[2]https://huggingface.co/spaces/bigcode/bigcode-models-leaderboard
[3]https://huggingface.co/bigcode/starcoder2-15b/discussions/6

[4]https://github.com/pschanely/CrossHair
[5]https://huggingface.co/codellama/CodeLlama-7b-hf

the targeted input with single and five attempts and then calculate the overall pass rates (see Section 3.2) respectively.

4.1.1 Overall Performance. The results of the four selected LLMs on both *logic_bombs* and *PathEval* dataset are shown in the Table 2. For clarity, we highlight the results in green and the degree of performance decline compared to *GPT-3.5* in red.

The results show that *GPT-3.5* achieves a significant and consistent performance advantage over the open-source LLMs under all subsets within our dataset. Specifically, *GPT-3.5* achieves the highest pass rate across all datasets under both a single attempt and multiple attempts. On *logic_bombs*, upon 43.4% inputs are generated correctly by *GPT -3.5*, and on *PathEval*, *GPT-3.5* also achieve the best results, ranging from 51.21% to 58.57% in all three different language datasets in *PathEval*. The performance gap between *GPT-3.5* and the open-source models is substantial, ranging from around 5.66% to over 36.02% in relative decline.

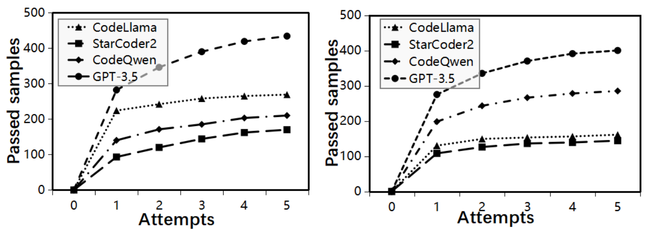
Finding 1: *GPT-3.5* exhibits excellent performance on directed input generation, passing 43.40%-58.57% samples in our dataset. For the studied open-source LLMs, the difference compared to *GPT-3.5* is relatively smaller in certain settings (less than 10%, minimum 5.66%). However, most of the time, there is a significant performance gap (more than 19%, maximum 36.02%).

4.1.2 Multilingual Perspective. *PathEval* dataset contains samples with semantically equivalent versions in multiple languages, which facilitates comparisons between LLMs from the perspective of different programming languages. The results are also shown in Table 2 distinct by the “Lang.” column.

We find that *GPT-3.5* also outperforms the open-source models across the multilingual dataset. However, upon closer inspection of the data for the three languages in *PathEval*, its generation quality for Java is slightly inferior to the other two languages, and the gap becomes more evident with only a single attempt. Among the open-source models, *StarCoder2* performs the best in C++ tasks, then on Python, and worst on Java. The other two models show obvious weaknesses in a certain language: the performance of *CodeQwen* in C++ is noticeably lower, around 75% (28.34% relative to 38.15% and 37.10%) of its performance when generating in the other two languages, yet the decline of *CodeLlama* in Java is even more severe, only about half (16.38% relative to 34.15% and 36.30%) of its performance on the other two languages. This discrepancy may be attributable to the varying understanding capabilities exhibited by the LLMs across different programming languages [64].

Finging 2: The performance of LLMs varies across languages. Specifically, the performance of *CodeLlama* is approximately halved on the Java samples compared to compared to its performance on C++ and Python datasets.

4.1.3 Multiple Attempts Improvement. From the “Att.” column in Table 2, we find that the multiple attempts strategy can greatly increase the pass rate for all the models, between 6.07% to 20.51%, varying from LLMs. We then further analyze the result within different numbers of attempts to understand trends in performance improvement. Figure 4a shows the number of pass samples under one to five attempts in the C++ subset of *PathEval* while Figure 4b shows the result of the Python subset. It is clear that in all passed samples, about 90% get correct input within three attempts for all LLMs, and over 95% in four attempts (relative to the result of five attempts). The improvement of *GPT-3.5* is higher than the 7b open-source LLMs in multiple attempts. In C++, there are 20.52% new success for 5 attempts than a single (9.45%, 10.39%, and 6.07% for *CodeQwen*, *StarCoder2* and *CodeLlama*) while in Python, the corresponding data are 16.73% (11.65%, 4.82% and 4.15% for *CodeQwen*, *StarCoder2* and *CodeLlama*). The divergent growth trends stem from GPT-3.5’s advantages in scale and training data, enabling it to generate valid targeted inputs with a higher chance.



(a) Result on *PathEval (C++)* **(b) Result on *PathEval (Python)***

Figure 4: Results for LLMs on *PathEval* with multiple attempts. Figure 4a shows the number of pass samples under 1-5 attempts on the C++ subset, while Figure 4b shows the result on the Python subset.

Table 3: The performances (i.e., pass rate) of constraint-based tools, *GPT-3.5* and *CodeLlama*. “GPT”, “CL”, “CH”, and “ES” represent *GPT-3.5*, *CodeLlama*, *CrossHair* and *EvoSuite*. Both the result of *GPT-3.5* and *CodeLlama* are in five attempts. "-" means the tool does not support the language.

Dataset	Lang.	*GPT (%)*	*CL (%)*	*Angr (%)*	*KLEE (%)*	*CH (%)*	*ES (%)*
logic-bombs	C	43.40	15.09	28.30	24.53	-	-
	C++	58.57	36.30	-	59.51	-	-
PathEval	Python	53.55	34.15	-	-	32.53	-
	Java	51.21	16.38	-	-	-	39.25

Finding 3: Multiple attempts are effective in increasing the performance for all LLMs (6.07% - 20.52%) while widening the gap between *GPT-3.5* and the smaller open-source LLMs.

4.2 RQ2: Comparison with Traditional Tools

Impressed by the excellent performance of LLMs in RQ1, we further explore the capabilities of the constraint-based tools, *KLEE*, *Angr*, *CrossHair* and *EvoSuite (in dynamic symbolic execution mode)*, on our benchmark, comparing with the performance of LLMs.

4.2.1 Overall Performance Comparison. The results of constraint-based tools are shown in Table 3. The results demonstrate that the performance of *GPT-3.5* on our datasets is on par with the constraint-based tools. On the Python and Java subsets of the *PathEval*, *GPT-3.5*

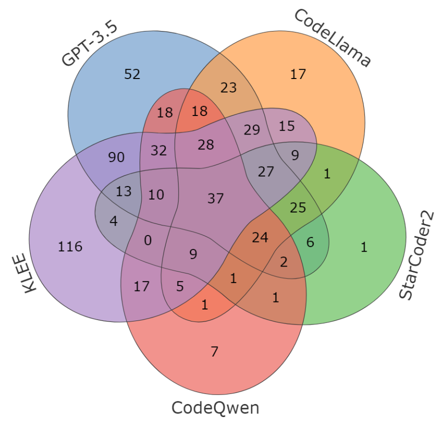


Figure 5: Venn plots of the samples passed by LLMs and *KLEE* on the C++ subset.

significantly outperforms the traditional tools that utilize dynamic symbolic execution techniques. When comparing to the highly mature symbolic execution tool *KLEE*, which has 2.5K stars on GitHub with over 2,700 commits, *GPT-3.5* is only inferior to it by 0.94% (59.51%-58.57%) on *PathEval (C++)*. Furthermore, on the *logic_bombs* dataset, which poses challenges for symbolic execution techniques, *GPT-3.5* can outperform *KLEE* and *Angr* by 18.87% (43.40%-24.53%) and 15.10% (43.40%-28.30%) respectively.

> **Finding 4**: *GPT-3.5* performs competitively to the engineering-proven constraint-based tools *KLEE* on our dataset, and significantly ahead of *CrossHair* and *EvoSuite* on Python and Java.

But are LLMs such as ChatGPT a panacea? The answer is probably **No**. We have taken a closer look at the passed samples in the *PathEval* dataset. Figure 5 illustrates the overlaps and differences in the number of them on *PathEval (C++)* dataset between LLMs and the *KLEE*. *KLEE* has the most unique successful samples, reaching 116, which accounts for 15.7% of the total samples. *GPT-3.5* and *CodeLlama* follow, accounting for 7.02% and 2.29% respectively. This strongly implicit that symbolic execution tools and LLMs may have their respective strengths and weaknesses, thus leading to a significant difference in their results. With this speculation in mind, we continue with a more detailed analysis from different perspectives (i.e., different scenarios of code, different forms of the input to be generated, and their generalization and comprehensiveness).

4.2.2 Code Scenario Perspective. First, intuitively, we categorize the samples by different code scenarios in order to explore the performance of each of the two types of tools in different situations. However, it is difficult to manually categorize more than 2,000 samples in a detailed and accurate manner. Therefore, we utilize samples in *logic_bombs*, which are already categorized based on the challenge scenarios for symbolic execution techniques when constructed artificially, to continue our study. For example, samples in the "cover_propagation" category denote there are disconnections in the propagation of constraints during the exploration of

Table 4: The categorized result of *GPT-3.5*, *KLEE* and *Angr* on *logic_bombs* dataset.

Categeroy	*GPT-3.5*	*KLEE*	*Angr*	*Total*
buffer_overflow	0	**2**	**2**	4
contextual_symbolic_value	**2**	**2**	0	5
covert_propogation	**6**	1	2	7
crypto_functions	0	0	0	2
external_functions	**5**	0	2	8
floating_point	**3**	0	2	5
integer_overflow	1	**2**	**2**	2
loop	**1**	0	0	5
parallel_program	**2**	0	0	5
symbolic_jump	0	**1**	0	3
symbolic_memory	3	**5**	4	7
Total	**23**	13	14	53

```
1  int logic_bomb(char* s) {
2      int pid, fd[2];
3      pipe(fd);
4      if ((pid = fork()) == -1) return 0;
5      if (pid == 0) {
6          close(fd[0]);
7          write(fd[1], s, sizeof(s));
8          wait(NULL);
9          exit(0);
10     } else {
11         char content[8];
12         close(fd[1]);
13         read(fd[0], content, 8);
14         if (strcmp(content, "7") == 0) {
15             // target line
16         }
17     }
18 }
```

Figure 6: An example in "parallel_program" that *KLEE* and *Angr* cannot model while all the LLMs pass.

the symbolic techniques (e.g., interaction with the operating system) in them while the category of "symbolic_memory" refers to the samples are related to memory operations (e.g., memory indexing and allocation). The results of different categories can highlight the strengths and limitations of constraint-based tools. Likewise, it allows us to better compare the advantages and disadvantages of LLMs and determine whether they align with those of the constraint-based approaches.

Table 4 shows the results of *GPT-3.5* (best in studied LLMs), and the two constraint-based tools with C language support, *KLEE*, and *Angr* on all the 11 categories in *logic_bombs*. We find that they have unique pros and cons. Among the 11 categories, *GPT-3.5* outperforms the two symbolic execution tools in five categories, while *KLEE* and *Angr* perform better in the other three categories.

Figure 6 showcases an example in the "parallel_program" category that demonstrates the advantages of LLMs. There are two processes (created by `fork` at L_4) that communicate with each other through pipes (L_3) in the program. Such implicit data flow is not modeled in *KLEE* or *Angr*, making it unprocessable by these tools. However, LLMs such as *GPT-3.5* and *CodeQwen*, can recognize the widely used parallel programming patterns, effectively understand the code and provide correct answers. Similar examples are widespread in categories like "external_functions" and

```c
int logic_bomb(char* symvar) {
    int flag = 0;
    char buf[8];
    if(strlen(symvar) > 9) return 0;
    strcpy(buf, symvar);
    if(flag == 1){
        // target line
    }
}
```

Figure 7: An example in "buffer_overflow" that *KLEE* and *Angr* can pass while all the LLMs cannot.

Table 5: Influence of pass rate by input/output features on *PathEval (C++)*. "GPT", "CL", "SC", and "CQ" refer to *GPT-3.5*, *CodeLlama*, *StarCoder2* and *CodeQwen*. "Str.", "Float." and "Aggr." refer to samples with string, floating point, and aggregate types of input/output. Data with large fluctuations (±10%) relative to the overall result (last row) are highlighted.

Features		*GPT (%)*	*CL (%)*	*SC (%)*	*CQ (%)*	*KLEE (%)*
Input	Output					
Str.	Any	53.65	36.91	24.46	27.90	59.23
Any	Str.	56.07	37.38	**19.16**	**22.43**	56.07
Float.	Any	56.52	33.33	21.74	**13.04**	**11.59**
Any	Float.	60.00	**40.00**	22.50	**15.00**	**7.50**
Aggr.	Aggr.	**66.03**	**43.59**	**27.56**	**23.72**	**10.26**
Aggr.	No Aggr.	**50.32**	**26.75**	**17.20**	**10.83**	54.78
No Aggr.	Aggr.	63.16	36.84	**20.18**	**38.60**	**69.30**
No Aggr.	No Aggr.	57.32	37.26	24.52	**35.67**	**82.80**
Any	Any	58.57	36.30	22.94	28.34	59.51

"covert_propagation", in which there are samples related to syscalls, external function calls, and constraints that are out of the source code (e.g., the input should be a valid IP address string). In addition, *KLEE* lacks support for symbolic floating-point computation without extensions, so the result is 0 in the "floating_point" category.

On the other hand, Figure 7 shows the strengths of constraint-based tools. In this stack overflow example, the input should be a string with the constraint of length equal to nine and ends with the specified character `\x01` (e.g. `"AAAAAAAA\x01"`). Such stringent strings can fulfill the `buf` array precisely by the first eight characters and overwrite the lowest 8 bits of variable `flag` next to `buf` on the stack to 1, finally reach to the target. Based on our observations, LLMs can understand the requirement of generating a long string to trigger an overflow. However, they struggle to control the exact length of the generated string and to produce specific characters for precise overwriting. As a result, none of the LLMs, including *GPT-3.5*, succeeded in this example. Similarly, issues arise with samples in the "integer overflow" and "symbolic jumps", both require precise calculations that *GPT-3.5* does not handle well.

Finding 5: LLMs can more effectively address challenges that involve implicit data flow (e.g., interactive with operation system) and out-of-code constraints (e.g., a valid IP address). However, their performance is less satisfactory when requiring precise solutions (e.g., an exact N-length string).

```cpp
vector<string> sorted_list_sum(vector<string> lst){
    vector<string> out={};
    for(int i=0;i<lst.size();i++)
        if(lst[i].length()%2==0) out.push_back(lst[i]);
    string mid;
    sort(out.begin(),out.end());
    for(int i=0;i<out.size();i++)
        for(int j=1;j<out.size();j++)
            if(out[j].length()<out[j-1].length()){
                mid=out[j];
                out[j]=out[j-1];
                out[j-1]=mid;}
    return out;
}
```

Figure 8: An example with aggregate input and output types that *KLEE* cannot pass any samples while LLMs pass all.

4.2.3 Input/Output Type. Additionally, we categorize the samples based on the characteristics of inputs and outputs. We specifically focus on the samples with non-basic types (i.e., floating point and string) and aggregate types (e.g., vector), which we recognize as widely used and more challenging. The results are shown in Table 5.

We observe more performance fluctuation for *KLEE*, *StarCoder2* and *CodeQwen* compared to *GPT-3.5* and *CodeLlama*. The performance of *GPT-3.5* and *CodeLlama* are significantly more stable. Except for the drop of floating pointer solving mentioned above, we find it interesting that when it comes to aggregate types, LLMs and *KLEE* perform differently. *KLEE* experienced a significant drop in performance when solving aggregate inputs and aggregate output samples; however, no significant performance degradation when dealing with aggregate inputs and not aggregate outputs. In contrast *GPT-3.5*, *CodeLlama* and *StarCoder2* behave oppositely.

We conjecture this could be the conversion between aggregate types involving a significant amount of complex constraints that are difficult to model effectively in practice. Figure 8 shows such an example that includes complex data and control flow in sorting. These are common in the conversion between vectors and other aggregate types and significantly challenge the symbolic execution technique. Additionally, the number of members in aggregate inputs is difficult to determine when declaring symbolic variables ahead of time. However, LLMs can generate code at a higher level of abstraction, avoiding the pitfalls of the underlying complex logic. On the other hand, the conversion from aggregate types to simple types often involves statistical operations such as the summation or selection of a set of data, which is a weakness for LLMs.

We also notice that when the samples of the input are not aggregate, *KLEE* performs excellently (82.80%); it also demonstrates the effectiveness of symbolic execution in ideal situations and the severity of the limitations as mentioned above in the real world.

Finding 6: LLMs and *KLEE* exhibit stark contrasts in handling aggregate inputs and outputs, with LLMs performing worse on the summation or selection elements from aggregate inputs while *KLEE* performing worse on aggregate type to aggregate type transformation such as array sorting.

We do not conduct the same analysis on the Python dataset due to the absence of type annotations in a part of the samples. However,

Table 6: The results of LLMs ("GPT", "CL", "SC", and "CQ" refer to *GPT- 3.5*, *CodeLlama*, textitStarCoder2, *CodeQwen*), and *CrossHair*, categorized based on whether there are parameter type annotations in the samples.

Type Annotation	*GPT (%)*	*CL (%)*	*SC (%)*	*CQ (%)*	*CrossHair (%)*
With	61.01	33.33	18.81	38.07	**45.41**
Without	50.47	34.62	19.47	38.19	**27.22**
Total	53.55	34.15	19.28	38.15	32.53

CrossHair requires the target functions to have type annotations. Otherwise, based on our observations, *CrossHair* always assumes that the input is of string type. To clarify the impact of this problem, we categorize the samples in *PathEval (Python)* based on whether the input parameters are typed. The results are shown in the Table 6.

Overall, we find that the absence of type annotations significantly reduces the effectiveness of *CrossHair* while having a smaller impact on *GPT-3.5* and *CodeLlama* since LLMs can infer type with a high precision [42]. In the category of samples without type annotations, *CrossHair*'s pass rate decreased dramatically by 40.06% ((45.51-27.22)/45.51) compared to the samples with that, while *GPT-3.5* only experienced a decrease of 17.56% ((61.01-50.47)/61.01), still at a relative high (50.47%) pass rate. Furthermore, there are even performance improvements for open-source LLMs (possibly due to the low coverage of type annotations in the open-source python projects [24] that are used as the primary training data).

Finding 7: LLMs have the potential to complete the missing information or manual work required by traditional tools and increase the overall level of automation (e.g., automatically generate symbolic value define for *KLEE* and type annotations for *CrossHair*) and performance of them.

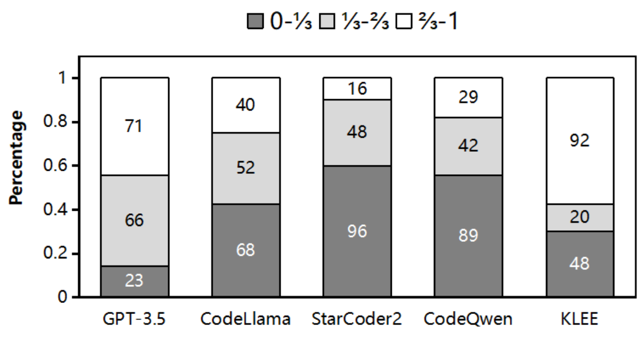


Figure 9: Numbers and percentage (relative to the total number of code groups) of LLMs and *KLEE* that passes less than 1/3, 1/3 to 2/3, and at least 2/3 samples in each code group.

4.2.4 Generalizability and Comprehensiveness. Furthermore, as test generation tools, the adaptability to diverse code and comprehensive test case generation is of paramount importance. Here we group the samples with the same code but different targets as a "code group" and calculate the pass rate of each code group. Figure 9 shows the numbers and percentage of each tool that passes less than 1/3, 1/3 to 2/3, and at least 2/3 samples in each code group (e.g. *GPT-3.5* has passed at least 2/3 samples in 71 code groups). Especially we also list the worst (pass rate is 0%, none of the samples in a code group is passed) and best (pass rate is 100%, all samples are passed in the code group) situations in Table 7.

Table 7: Numbers and percentage of different code groups that samples in the groups are all passed or all failed.

Categeroy	*GPT-3.5*		*CodeLlama*		*StarCoder2*		*CodeQwen*		*KLEE*	
All Fail ↓	**10**	**6.25%**	24	15.00%	58	36.25%	56	35.00%	37	23.13%
All Pass ↑	45	28.13%	16	10.00%	9	5.63%	18	11.25%	**80**	**50.00%**

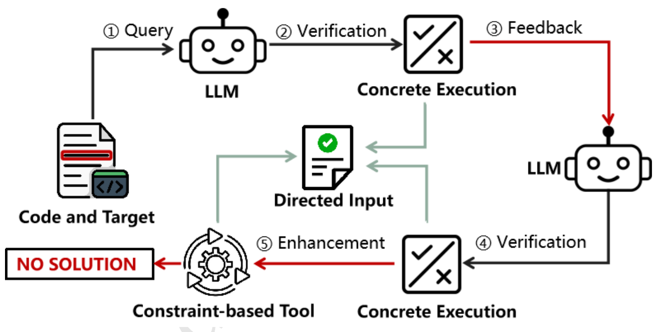


Figure 10: Overview of *LLMSym*.

We find a significant difference in the distributions between *KLEE* and LLMs. *KLEE* has a higher proportion in the "All Pass" code groups shown in Table 7, while *GPT-3.5* and *CodeLlama* have a lower proportion in "All Fail" groups. This indicates that when dealing with diverse code, *GPT-3.5* is more likely to provide *some* valid inputs and is rarely unavailable. However, on the other hand, the comprehensiveness of the test cases generated by *GPT-3.5* is not as robust as those generated by *KLEE*. *KLEE* can fully solve 50% of the code groups, but a significant portion of the groups remains entirely unsolvable. In these cases, symbolic execution processing in *KLEE* may encounter critical limitations such as path explosion, external interactions, and nonlinear constraints.

Finding 8: *GPT-3.5* have better generalizability as it can often provide some valid inputs for a program, but its ability to generate comprehensive test cases is not as strong as *KLEE*. In contrast, *KLEE* is more code-sensitive. It can solve all paths on half of the code snippets in our dataset. However, there are also 28.13% samples that do not find any solution at all.

4.3 RQ3: Integrating LLMs with Traditional Tools for Better Performance

Considering the unique pros and cons of LLMs and traditional tools in directed input generation, we design an approach combining the advantages of LLM and traditional tools as shown in Figure 10, named ***LLMSym***. The key insight of *LLMSym* is to take the advantages of both LLM and traditional tools. The key workflow and the corresponding insights of *LLMSym* are as follows.

Given the excellent pass rate (Finding 1, 2, 4) and stability (Finding 8) of *GPT-3.5*, we will first query LLMs to generate candidate solutions (i.e., directed inputs) in step ① and verify the candidates via concrete execution one by one in step ②. If a candidate passes the validation, it is returned as a solution, and the process ends. If none of the candidates pass, then we return the feedback (i.e., the actual output of the LLM-generated inputs produced by concrete execution) to LLMs for further refinement in step ③. Then, the refined solutions are, again, verified via concrete execution in Step ④. Finally, if LLMs still cannot generate solutions, we consider this a problem that LLMs are not suited to solve, such as precise calculations (Finding 5, 6), we thus turn to traditional tools. Step ⑤ enhances the harness, we extract useful features from the previous LLMs generated inputs (i.e. aggregate data structure sizes and input types) to construct more tailored harnesses (Finding 7).

Beyond direct integration, two key novel designs in the above workflow are as follows. First, we leverage the feedback shot in step ③ to maximize the gains from multiple attempts, which we find useful in Finding 3. By feedback with problematic generated inputs and execution-earned outputs, we aim to prohibit LLMs from making the same mistakes in subsequent generations [16]. Second, we enhanced the harness of constraint-base tools in step ⑤. At a high level, it can be viewed as utilizing LLMs-generated inputs to guide the symbolic value defined in constraint-based tools. For example, if LLMs generate `[1,2]` as input, we preserve the structure of the inputs and replace the concrete values with symbolic ones (i.e., `[i0, i1]`, `i0` and `i1` are symbolic integer values) to generate harnesses. Instead of a fixed template, we utilize the code-aware output from LLMs to generate more tailored and flexible harnesses for the constraint-based tools as we find that LLMs can infer more information from the code, as revealed in Finding 7.

We further implement the pipeline in Figure 10 based on *GPT-3.5*, *KLEE* and *CrossHair*. The experiment is conducted on *PathEval* C++ and Python datasets. The result is shown in Table 8. In particular, for Python, the overall pass rate is 74.43% (556/747), outperforming the baselines *GPT-3.5* by 39.00% ((556-400)/400) and *CrossHair* by 128.81% ((556-243)/243), Notably, there are 74 new successes that both baselines fail to cover. While for C++, the pass rate reaches 85.83% (636/741), outperforming *KLEE* by 44.22% ((636-441)/441) and *GPT-3.5* by 46.54% ((636-434)/434). And there are 28 new successes that both *KLEE* and *GPT-3.5* had previously failed to achieve.

We also investigate the impact of feedback shot in step ③ and harness enhancement in step ⑤ via ablation study. Results are shown in Table 8 under the *LLMSym* entry. Compared with the results achieved by full LLMSym (556 and 636 on two datasets), excluding these two designs causes obvious drops. The lack of the feedback shot led to 48 failed Python samples and 21 failed C++ samples compared to the full version. Likewise, the absence of the harness enhancement resulted in 23 failed Python samples and 5 failed C++ samples. Removing both components led to a combined total of 74 failed Python samples and 28 C++ samples.

Table 8: Numbers of passed samples for baselines and *LLMSym*. "Full" indicates the result of the complete *LLMSym* while "w/o ③", "w/o ⑤" and "w/o ③⑤" indicate the results without feedback shot prompt, without the enhanced harness, and without both respectively.

Dataset	Total	Baseline			LLMSym			
PathEval		*GPT*	*CrossHair*	*KLEE*	Full	w/o ③	w/o ⑤	w/o ③⑤
Python	747	400	243	-	**556**	508	533	482
C++	741	434	-	441	**636**	615	631	608

Finding 9: It is feasible and promising to integrate LLMs with constraint-based tools and evaluation results show that our novel solution can yield significant improvements. We believe this is a promising direction worthy of further exploration.

5 Threat to Validity

This paper has two main threats to validity.

First, The findings may not be ***generalizable*** to other LLMs and tools. To alleviate this threat, for LLMs, we selected representative ones following previous works [25, 31]. For traditional tools, we selected the mainstream commercialized symbolic/concolic execution tools following prior work [60, 61] and avoided those [19, 39] suffer from practical limitations (e.g., the precise construction of cross-function control flow graphs) as pointed out by prior study [6].

Second, the prompt we used may not be ***optimal***. To alleviate this threat, we designed the prompt template following the principle of prior work [38]. Though advanced prompting techniques (e.g., in-context learning or Chain-of-Thought) and more sophisticated prompting techniques from recent test generation work (e.g., prompt with the results of static analysis) may yield better results, which is remote from the core of this paper. Furthermore, the hybrid method we propose in our paper is orthogonal prompt engineering and can be easily integrated with those prompt-related advancements. So we leave it to further exploration.

6 Related Work

Directed test input using LLMs. Some LLMs test case generation works utilize specialized prompts to create directed test input that covers code branches and areas not reached and further improve the overall coverage, mutation scores, and other metrics. *TestGen-LLM* [2] introduces the engineering practice in Meta, they prompt the LLMs to generate tests that can increase the test coverage or cover missing corner cases in the instruction directly. *CoverUp* [43] iteratively improves coverage by adding coverage information to prompts to make LLMs focus their attention on as yet uncovered lines and branches. *MuTAP* [16] highlight the undetected mutation position in the prompt and ask LLMs to generate a new test case to cover it. *SymPrompt* [47] conducts a lightweight analysis on the program under test to extract the constraints of each path, then generate a path-specific prompt with constraints to generate more complete test cases. *TELPA* [62] integrates static analysis results and corner cases into prompts to guide LLMs to generate diverse tests that reach hard-to-cover branches. However, none of these works have thoroughly explored the perspective of combining constraint-based tools to mitigate the limitations of LLMs.

Comparison or integration between LLMs and traditional tools. The emergence of LLMs has introduced new options for software engineering tasks. Consequently, some works focus on comparison between Search-Based Software Testing (SBST) tools with new LLMs tools to gain a fuller understanding of LLMs or better performance. Tang et al. [52] present a systematic assessment of unit test suites generated by two state-of-the-art techniques: *ChatGPT* and SBST tools. They comprehensively evaluate test suites generated by *ChatGPT* from the perspectives of correctness, readability, code coverage, and bug detection capability and found that the state-of-the-art SBST tools gain higher code coverage than *ChatGPT*. Bhatia [7] conducted an experimental investigation to compare the quality of unit tests generated by LLMs against those generated by the commonly used test generator *Pynguin* [36]. Furthermore, *Codamosa* [30] is a system that integrates queries to LLMs into search-based algorithms for unit test generation. *Codamosa* conducts SBST until coverage improvements stall, then asks LLM to provide example test cases for under-covered functions, which help SBST redirect its search to more useful areas. However, to the best of our knowledge, there are few investigations that compare and integrate LLMs with constraint-based tools on test case generation.

7 Conclusion

In this article, we present a systematic study of the performance of LLMs on directed test input generations. We conduct the first comparative study between symbolic/concolic execution tools and LLMs on this task to reveal the differences from multiple critical perspectives, including the impact of different programming languages, code scenarios, features of inputs, the sensitivity of code, etc. We find that a) LLMs such as *ChatGPT* are comparable to or even better than the constraint-based tools. b) However, the difference between LLMs and traditional tools is also significant. They have distinct advantages and disadvantages from different perspectives. c) Based on our findings, we propose a simple but effective approach to combining their unique strengths. The evaluation result shows that our approach can significantly outperform the baselines.

Acknowledgments

We sincerely thank all anonymous reviewers for their valuable feedback and guidance in improving this paper. This work was sponsored by the Major Program (JD) of Hubei Province (No.2023BAA024).

References

[1] Muhammad Azeem Akbar, Arif Ali Khan, and Peng Liang. 2023. Ethical Aspects of ChatGPT in Software Engineering Research. *CoRR* abs/2306.07557 (2023). https://doi.org/10.48550/ARXIV.2306.07557 arXiv:2306.07557

[2] Nadia Alshahwan, Jubin Chheda, Anastasia Finegenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. 2024. Automated Unit Test Improvement using Large Language Models at Meta. *CoRR* abs/2402.09171 (2024). https://doi.org/10.48550/ARXIV.2402.09171 arXiv:2402.09171

[3] Andrea Arcuri, Juan Pablo Galeotti, Bogdan Marculescu, and Man Zhang. 2021. EvoMaster: A Search-Based System Test Generation Tool. *J. Open Source Softw.* 6, 57 (2021), 2153. https://doi.org/10.21105/JOSS.02153

[4] Shay Artzi, Julian Dolby, Frank Tip, and Marco Pistoia. 2010. Directed test generation for effective fault localization. In *Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010, Trento, Italy, July 12-16, 2010*, Paolo Tonella and Alessandro Orso (Eds.). ACM, 49–60. https://doi.org/10.1145/1831708.1831715

[5] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. 2023. Qwen technical report. *arXiv preprint arXiv:2309.16609* (2023).

[6] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.* 51, 3 (2018), 50:1–50:39. https://doi.org/10.1145/3182657

[7] Shreya Bhatia, Tarushi Gandhi, Dhruv Kumar, and Pankaj Jalote. 2023. Unit Test Generation using Generative AI : A Comparative Performance Analysis of Autogeneration Tools. *CoRR* abs/2312.10622 (2023). https://doi.org/10.48550/ARXIV.2312.10622 arXiv:2312.10622

[8] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 2329–2344. https://doi.org/10.1145/3133956.3134020

[9] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, Richard Draves and Robbert van Renesse (Eds.). USENIX Association, 209–224. http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf

[10] Sadullah Canakci, Nikolay Matyunin, Kalman Graffi, Ajay Joshi, and Manuel Egele. 2022. TargetFuzz: Using DARTs to Guide Directed Greybox Fuzzers. In *ASIA CCS '22: ACM Asia Conference on Computer and Communications Security, Nagasaki, Japan, 30 May 2022 - 3 June 2022*, Yuji Suga, Kouichi Sakurai, Xuhua Ding, and Kazue Sako (Eds.). ACM, 561–573. https://doi.org/10.1145/3488932.3501276

[11] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2023. CodeT: Code Generation with Generated Tests. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net. https://openreview.net/pdf?id=ktrw68Cmu9c

[12] Eason Chen, Ray Huang, Han-Shin Chen, Yuen-Hsien Tseng, and Liang-Yi Li. 2023. GPTutor: A ChatGPT-Powered Programming Tool for Code Explanation. In *Artificial Intelligence in Education. Posters and Late Breaking Results, Workshops and Tutorials, Industry and Innovation Tracks, Practitioners, Doctoral Consortium and Blue Sky - 24th International Conference, AIED 2023, Tokyo, Japan, July 3-7, 2023, Proceedings (Communications in Computer and Information Science, Vol. 1831)*, Ning Wang, Genaro Rebolledo-Mendez, Vania Dimitrova, Noboru Matsuda, and Olga C. Santos (Eds.). Springer, 321–327. https://doi.org/10.1007/978-3-031-36336-8_50

[13] Jiawei Chen, Hongyu Lin, Xianpei Han, and Le Sun. 2024. Benchmarking Large Language Models in Retrieval-Augmented Generation. In *Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2024, Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence, IAAI 2024, Fourteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2014, February 20-27, 2024, Vancouver, Canada*, Michael J. Wooldridge, Jennifer G. Dy, and Sriraam Natarajan (Eds.). AAAI Press, 17754–17762. https://doi.org/10.1609/AAAI.V38I16.29728

[14] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, et al. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021). arXiv:2107.03374 https://arxiv.org/abs/2107.03374

[15] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, Martin Odersky and Philip Wadler (Eds.). ACM, 268–279. https://doi.org/10.1145/351240.351266

[16] Arghavan Moradi Dakhel, Amin Nikanjam, Vahid Majdinasab, Foutse Khomh, and Michel C. Desmarais. 2023. Effective Test Generation Using Pre-trained Large Language Models and Mutation Testing. *CoRR* abs/2308.16557 (2023). https://doi.org/10.48550/ARXIV.2308.16557 arXiv:2308.16557

[17] Marian Daun and Jennifer Brings. 2023. How ChatGPT Will Change Software Engineering Education. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1, ITiCSE 2023, Turku, Finland, July 7-12, 2023*, Mikko-Jussi Laakso, Mattia Monga, Simon, and Judithe Sheard (Eds.). ACM, 110–116. https://doi.org/10.1145/3587102.3588815

[18] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large Language Models Are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via Large Language Models. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, René Just and Gordon Fraser (Eds.). ACM, 423–435. https://doi.org/10.1145/3597926.3598067

[19] Peter Dinges and Gul A. Agha. 2014. Targeted test input generation using symbolic-concrete backward execution. In *ACM/IEEE International Conference on*

Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014, Ivica Crnkovic, Marsha Chechik, and Paul Grünbacher (Eds.). ACM, 31–36. https://doi.org/10.1145/2642937.2642951

[20] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated Repair of Programs from Large Language Models. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 1469–1481. https://doi.org/10.1109/ICSE48619.2023.00128

[21] Andrea Fioraldi, Dominik Christian Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ : Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies, WOOT 2020, August 11, 2020*, Yuval Yarom and Sarah Zennou (Eds.). USENIX Association. https://www.usenix.org/conference/woot20/presentation/fioraldi

[22] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. In *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, Tibor Gyimóthy and Andreas Zeller (Eds.). ACM, 416–419. https://doi.org/10.1145/2025113.2025179

[23] Gordon Fraser and Andrea Arcuri. 2013. EvoSuite: On the Challenges of Test Case Generation in the Real World. In *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, 2013*. IEEE Computer Society, 362–369. https://doi.org/10.1109/ICST.2013.51

[24] Luca Di Grazia and Michael Pradel. 2022. The evolution of type annotations in python: an empirical study. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, Abhik Roychoudhury, Cristian Cadar, and Miryung Kim (Eds.). ACM, 209–220. https://doi.org/10.1145/3540250.3549114

[25] Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I. Wang. 2024. CRUXEval: A Benchmark for Code Reasoning, Understanding and Execution. *CoRR* abs/2401.03065 (2024). https://doi.org/10.48550/ARXIV.2401.03065 arXiv:2401.03065

[26] Qi Guo, Junming Cao, Xiaofei Xie, Shangqing Liu, Xiaohong Li, Bihuan Chen, and Xin Peng. 2024. Exploring the Potential of ChatGPT in Automated Code Refinement: An Empirical Study. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 34:1–34:13. https://doi.org/10.1145/3597503.3623306

[27] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John C. Grundy, and Haoyu Wang. 2023. Large Language Models for Software Engineering: A Systematic Literature Review. *CoRR* abs/2308.10620 (2023). https://doi.org/10.48550/ARXIV.2308.10620 arXiv:2308.10620

[28] Sihao Hu, Tiansheng Huang, Fatih Ilhan, Selim Furkan Tekin, and Ling Liu. 2023. Large Language Model-Powered Smart Contract Vulnerability Detection: New Perspectives. In *5th IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications, TPS-ISA 2023, Atlanta, GA, USA, November 1-4, 2023*. IEEE, 297–306. https://doi.org/10.1109/TPS-ISA58951.2023.00044

[29] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of Code Language Models on Automated Program Repair. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 1430–1442. https://doi.org/10.1109/ICSE48619.2023.00125

[30] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. 2023. CodaMosa: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 919–931. https://doi.org/10.1109/ICSE48619.2023.00085

[31] Jia Li, Ge Li, Xuanming Zhang, Yihong Dong, and Zhi Jin. 2024. EvoCodeBench: An Evolving Code Generation Benchmark Aligned with Real-World Code Repositories. *CoRR* abs/2404.00599 (2024). https://doi.org/10.48550/ARXIV.2404.00599 arXiv:2404.00599

[32] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, et al. 2023. StarCoder: may the source be with you! *CoRR* abs/2305.06161 (2023). https://doi.org/10.48550/ARXIV.2305.06161 arXiv:2305.06161

[33] Hongliang Liang, Lin Jiang, Lu Ai, and Jinyi Wei. 2020. Sequence Directed Hybrid Fuzzing. In *27th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2020, London, ON, Canada, February 18-21, 2020*, Kostas Kontogiannis, Foutse Khomh, Alexander Chatzigeorgiou, Marios-Eleftherios Fokaefs, and Minghui Zhou (Eds.). IEEE, 127–137. https://doi.org/10.1109/SANER48275.2020.9054807

[34] Andreas Löscher and Konstantinos Sagonas. 2017. Targeted property-based testing. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*, Tevfik Bultan and Koushik Sen (Eds.). ACM, 46–56. https://doi.org/10.1145/3092703.3092711

[35] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, and et al. 2024. StarCoder 2 and The Stack v2: The Next Generation. *CoRR* abs/2402.19173 (2024). https://doi.org/10.48550/ARXIV.2402.19173 arXiv:2402.19173

[36] Stephan Lukasczyk and Gordon Fraser. 2022. Pynguin: Automated Unit Test Generation for Python. In *44th IEEE/ACM International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2022, Pittsburgh, PA, USA, May 22-24, 2022*. ACM/IEEE, 168–172. https://doi.org/10.1145/3510454.3516829

[37] Changhua Luo, Wei Meng, and Penghui Li. 2023. SelectFuzz: Efficient Directed Fuzzing with Selective Path Exploration. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*. IEEE, 2693–2707. https://doi.org/10.1109/SP46215.2023.10179296

[38] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. WizardCoder: Empowering Code Large Language Models with Evol-Instruct. *CoRR* abs/2306.08568 (2023). https://doi.org/10.48550/ARXIV.2306.08568 arXiv:2306.08568

[39] Kin-Keung Ma, Yit Phang Khoo, Jeffrey S. Foster, and Michael Hicks. 2011. Directed Symbolic Execution. In *Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6887)*, Eran Yahav (Ed.). Springer, 95–111. https://doi.org/10.1007/978-3-642-23702-7_11

[40] Rupak Majumdar and Ru-Gang Xu. 2007. Directed test generation using symbolic grammars. In *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA*, R. E. Kurt Stirewalt, Alexander Egyed, and Bernd Fischer (Eds.). ACM, 134–143. https://doi.org/10.1145/1321631.1321653

[41] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. 2023. Examining Zero-Shot Vulnerability Repair with Large Language Models. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*. IEEE, 2339–2356. https://doi.org/10.1109/SP46215.2023.10179420

[42] Yun Peng, Chaozheng Wang, Wenxuan Wang, Cuiyun Gao, and Michael R. Lyu. 2023. Generative Type Inference for Python. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 988–999. https://doi.org/10.1109/ASE56229.2023.00031

[43] Juan Altmayer Pizzorno and Emery D. Berger. 2024. CoverUp: Coverage-Guided LLM-Based Test Generation. *CoRR* abs/2403.16218 (2024). https://doi.org/10.48550/ARXIV.2403.16218 arXiv:2403.16218

[44] Laura Plein, Wendkûuni C. Ouédraogo, Jacques Klein, and Tegawendé F. Bissyandé. 2024. Automatic Generation of Test Cases based on Bug Reports: a Feasibility Study with Large Language Models. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 360–361. https://doi.org/10.1145/3639478.3643119

[45] Moumita Das Purba, Arpita Ghosh, Benjamin J. Radford, and Bill Chu. 2023. Software Vulnerability Detection using Large Language Models. In *34th IEEE International Symposium on Software Reliability Engineering, ISSRE 2023 - Workshops, Florence, Italy, October 9-12, 2023*. IEEE, 112–119. https://doi.org/10.1109/ISSREW60843.2023.00058

[46] Nikitha Rao, Kush Jain, Uri Alon, Claire Le Goues, and Vincent J. Hellendoorn. 2023. CAT-LM Training Language Models on Aligned Code And Tests. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 409–420. https://doi.org/10.1109/ASE56229.2023.00193

[47] Gabriel Ryan, Siddhartha Jain, Mingyue Shang, Shiqi Wang, Xiaofei Ma, Murali Krishna Ramanathan, and Baishakhi Ray. 2024. Code-Aware Prompting: A study of Coverage Guided Test Generation in Regression Setting using LLM. *CoRR* abs/2402.00097 (2024). https://doi.org/10.48550/ARXIV.2402.00097 arXiv:2402.00097

[48] Sami Sarsa, Paul Denny, Arto Hellas, and Juho Leinonen. 2022. Automatic Generation of Programming Exercises and Code Explanations Using Large Language Models. In *ICER 2022: ACM Conference on International Computing Education Research, Lugano and Virtual Event, Switzerland, August 7 - 11, 2022, Volume 1*, Jan Vahrenhold, Kathi Fisler, Matthias Hauswirth, and Diana Franklin (Eds.). ACM, 27–43. https://doi.org/10.1145/3501385.3543957

[49] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2024. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. *IEEE Trans. Software Eng.* 50, 1 (2024), 85–105. https://doi.org/10.1109/TSE.2023.3334955

[50] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy.*

[51] Chang-Ai Sun, Baoli Liu, An Fu, Yiqiang Liu, and Huai Liu. 2022. Path-directed source test case generation and prioritization in metamorphic testing. *J. Syst. Softw.* 183 (2022), 111091. https://doi.org/10.1016/J.JSS.2021.111091

[52] Yutian Tang, Zhijie Liu, Zhichao Zhou, and Xiapu Luo. 2023. ChatGPT vs SBST: A Comparative Assessment of Unit Test Suite Generation. *CoRR* abs/2307.00588 (2023). https://doi.org/10.48550/ARXIV.2307.00588 arXiv:2307.00588

[53] Chandra Thapa, Seung Ick Jang, Muhammad Ejaz Ahmed, Seyit Camtepe, Josef Pieprzyk, and Surya Nepal. 2022. Transformer-Based Language Models for Software Vulnerability Detection. In *Annual Computer Security Applications*

Conference, ACSAC 2022, Austin, TX, USA, December 5-9, 2022. ACM, 481–496. https://doi.org/10.1145/3564625.3567985

[54] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, et al. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. *CoRR* abs/2307.09288 (2023). https://doi.org/10.48550/ARXIV.2307.09288 arXiv:2307.09288

[55] Vasudev Vikram, Caroline Lemieux, and Rohan Padhye. 2023. Can Large Language Models Write Good Property-Based Tests? *CoRR* abs/2307.04346 (2023). https://doi.org/10.48550/ARXIV.2307.04346 arXiv:2307.04346

[56] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software Testing With Large Language Models: Survey, Landscape, and Vision. *IEEE Trans. Software Eng.* 50, 4 (2024), 911–936. https://doi.org/10.1109/TSE.2024.3368208

[57] Jules White, Sam Hays, Quchen Fu, Jesse Spencer-Smith, and Douglas C. Schmidt. 2023. ChatGPT Prompt Patterns for Improving Code Quality, Refactoring, Requirements Elicitation, and Software Design. *CoRR* abs/2303.07839 (2023). https://doi.org/10.48550/ARXIV.2303.07839 arXiv:2303.07839

[58] 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. https://doi.org/10.1145/3597503.3639121

[59] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated Program Repair in the Era of Large Pre-trained Language Models. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023.* IEEE, 1482–1494. https://doi.org/10.1109/ICSE48619.2023.00129

[60] Hui Xu, Zirui Zhao, Yangfan Zhou, and Michael R. Lyu. 2020. Benchmarking the Capability of Symbolic Execution Tools with Logic Bombs. *IEEE Trans. Dependable Secur. Comput.* 17, 6 (2020), 1243–1256. https://doi.org/10.1109/TDSC.2018.2866469

[61] Hui Xu, Yangfan Zhou, Yu Kang, and Michael R. Lyu. 2017. Concolic Execution on Small-Size Binaries: Challenges and Empirical Study. In *47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2017, Denver, CO, USA, June 26-29, 2017.* IEEE Computer Society, 181–188. https://doi.org/10.1109/DSN.2017.11

[62] Chen Yang, Junjie Chen, Bin Lin, Jianyi Zhou, and Ziqi Wang. 2024. Enhancing LLM-based Test Generation for Hard-to-Cover Branches via Program Analysis. *CoRR* abs/2404.04966 (2024). https://doi.org/10.48550/ARXIV.2404.04966 arXiv:2404.04966

[63] Quanjun Zhang, Tongke Zhang, Juan Zhai, Chunrong Fang, Bowen Yu, Weisong Sun, and Zhenyu Chen. 2023. A Critical Review of Large Language Model on Software Engineering: An Example from ChatGPT and Automated Program Repair. *CoRR* abs/2310.08879 (2023). https://doi.org/10.48550/ARXIV.2310.08879 arXiv:2310.08879

[64] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023. CodeGeeX: A Pre-Trained Model for Code Generation with Multilingual Benchmarking on HumanEval-X. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, KDD 2023, Long Beach, CA, USA, August 6-10, 2023*, Ambuj K. Singh, Yizhou Sun, Leman Akoglu, Dimitrios Gunopulos, Xifeng Yan, Ravi Kumar, Fatma Ozcan, and Jieping Ye (Eds.). ACM, 5673–5684. https://doi.org/10.1145/3580305.3599790

[65] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. 2022. Fuzzing: A Survey for Roadmap. *ACM Comput. Surv.* 54, 11s (2022), 230:1–230:36. https://doi.org/10.1145/3512345