
# Unlocking Low Frequency Syscalls in Kernel Fuzzing with Dependency-Based RAG

ZHIYU ZHANG, IIE at CAS, China and UCAS, China
LONGXING LI, IIE at CAS, China and UCAS, China
RUIGANG LIANG*, IIE at CAS, China and UCAS, China
KAI CHEN*, IIE at CAS, China and UCAS, China

Most coverage-guided kernel fuzzers test operating system kernels based on syscall sequence synthesis. However, there are still syscalls rarely or not covered (called low frequency syscalls, LFS) in a period of fuzzing, meaning the relevant code branches remain unexplored. This is due to the complex dependencies of the LFS and mutation uncertainty, which makes it difficult for fuzzers to generate corresponding syscall sequences. Since many kernel fuzzers can dynamically learn syscall dependencies from the current corpus based on the choice table mechanism, providing comprehensive and high-quality seeds could help fuzzers cover LFS. However, constructing such seeds relies heavily on expert experience to resolve the syscall dependencies.

In this paper, we propose SyzGPT, the first kernel fuzzing framework to automatically generate effective seeds for LFS via Large Language Model (LLM). We leverage a dependency-based retrieval-augmented generation (DRAG) method to unlock the potential of LLM and design a series of steps to improve the effectiveness of the generated seeds. First, SyzGPT automatically extracts syscall dependencies from the existing documentation via LLM. Second, SyzGPT retrieves programs from the fuzzing corpus based on the dependencies to construct adaptive context for LLM. Last, SyzGPT periodically generates and repairs seeds with feedback to enrich the fuzzing corpus for LFS. We propose a novel set of evaluation metrics for seed generation in kernel domain. Our evaluation shows that SyzGPT can generate seeds with a high valid rate of 87.84% and can be extended to offline and fine-tuned LLMs. Compared to seven state-of-the-art kernel fuzzers, SyzGPT improves code coverage by 17.73%, LFS coverage by 58.00%, and vulnerability detection by 323.22% on average. Besides, SyzGPT independently discovered 26 unknown kernel bugs (10 are LFS-related), with 11 confirmed.

CCS Concepts: • **Security and privacy → Operating systems security**.

Additional Key Words and Phrases: Syscall Dependency, Kernel Fuzzing, RAG, Seed Generation

**ACM Reference Format:**
Zhiyu Zhang, Longxing Li, Ruigang Liang, and Kai Chen. 2025. Unlocking Low Frequency Syscalls in Kernel Fuzzing with Dependency-Based RAG. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA038 (July 2025), 23 pages. https://doi.org/10.1145/3728913

*Corresponding authors.
†Institute of Information Engineering at Chinese Academy of Sciences
‡University of Chinese Academy of Sciences

Authors' Contact Information: Zhiyu Zhang, State Key Laboratory of Cyberspace Security Defense, IIE at CAS , Beijing, China and UCAS , School of Cyber Security, Beijing, China, zhangzhiyu1999@iie.ac.cn; Longxing Li, State Key Laboratory of Cyberspace Security Defense, IIE at CAS, Beijing, China and UCAS, School of Cyber Security, Beijing, China, lilongxing@iie.ac.cn; Ruigang Liang, State Key Laboratory of Cyberspace Security Defense, IIE at CAS, Beijing, China and UCAS, School of Cyber Security, Beijing, China, liangruigang@iie.ac.cn; Kai Chen, State Key Laboratory of Cyberspace Security Defense, IIE at CAS, Beijing, China and UCAS, School of Cyber Security, Beijing, China, chenkai@iie.ac.cn.


ACM 2994-970X/2025/7-ARTISSTA038
https://doi.org/10.1145/3728913

## 1 Introduction

Kernel is the core component of the operating system that manages the resources and provides interfaces for other components. Security vulnerabilities in the kernel often have wide-ranging impacts and serious hazards. For example, Null-Pointer-Dereference can lead to memory leak or kernel panic [3], Out-Of-Bounds can result in denial of service or privilege escalation [2], and Use-After-Free can be exploited to privilege escalation or remote code execution [4]. According to CVE Details [53], hundreds of Linux kernel CVEs have been reported yearly, increasing the urgency of discovering and fixing kernel vulnerabilities. As the scale of operating systems continues to grow [1], discovering vulnerabilities through manual auditing becomes difficult and impractical. Fuzzing, an automated method for detecting bugs in software and systems, has proven particularly effective in various scenarios, including kernels. As the most widely used coverage-guided kernel fuzzer, Syzkaller [12] has found around 7k bugs in Linux kernel by Oct 2024. These modern kernel fuzzers generally test kernels by invoking sequences of system calls (*syscalls*), also known as syscall-based kernel fuzzers. The syscall sequences (*Syzkaller-style programs*[1]) are synthesized and mutated under the guidance of syscall description language (e.g., *Syzlang* [13]). Then, the programs are scheduled with specific guiding mechanisms, such as coverage-guided [28, 56], subsystem-guided [35, 67], and vulnerability-guided [32, 71], and persisted in *corpus* (a set of effective *programs*).

In general, the more valid syscalls synthesized in the corpus, the more code branches the fuzzer explores, leading to the discovery of more bugs. Syzlang describes over 4400 specialized calls[2] to different interfaces of more than 360 system calls defined in the Linux kernel. However, many calls take efforts to cover. According to our observations, the latest Syzkaller can only cover around 57% of the defined specialized calls (i.e., 2534 of 4446) on average in several 24-hour fuzzing tests. We call the rarely[3] or never covered syscalls *Low Frequency Syscalls* (LFS) and *Low Frequency Specialized sysCalls* (LFSC, which is the granularity of our research in practice). LFSC may exist due to environment dependencies, unresolved syscall dependencies [30], and the randomness in fuzzing. Among the approximately 1900 observed LFSC, around five hundred device-related specialized calls are automatically identified as disabled before fuzzing. Since there are efforts in environment emulation [48, 51] to solve the coverage of these syscalls, the rest of LFSC are our research scope.

**Motivation.** An intuitive solution to cover them is executing effective programs containing these LFSC during fuzzing. However, generating high-quality programs for the LFSC faces the following limitations. **L1:** *LFSC are hard to cover by fuzzer mutation.* The syscall-based fuzzers generally mutate the programs according to the dependencies they learn from execution (e.g., *choice table* in Syzkaller and *relation table* in Healer [56]). Since LFSC are rare or not even in the corpus, it is difficult for fuzzers to learn the corresponding dependencies of LFSC. Consequently, the fuzzers can only rely on the random generation to cover the LFSC, which is inefficient. **L2:** *The complex syscall dependencies are hard to extract automatically.* Constructing semantically valid programs for LFSC requires conforming to the syscall synopsis and satisfying syscall dependencies, which relies on expert experience. The usage and dependencies of the syscalls are described in the kernel documentation (e.g., *manpage* [7]) and *Syzlang* documentation. Some syscall dependencies are implicitly defined in kernel source code, which cannot be thoroughly and accurately extracted through static analysis. Previous studies attempted to automatically extract syscall dependencies by combining dynamic execution and static analysis [45, 56, 73]. However, it is time-consuming to learn all dependencies from execution gradually. As shown at the left of the motivating example in Figure 1, the limitations restrict the traditional fuzzers from reaching the LFSC-related bug entry.

[1]Syz-programs (also abbreviated as programs) are inputs of the kernel under test.
[2]Specialized calls are encapsulated for specific purposes of system calls, e.g., `ioctl$I2C_RDWR` is a specialized call of `ioctl`.
[3]In general, we consider syscalls that cannot be covered consistently over several 24-hour fuzzing as rare.

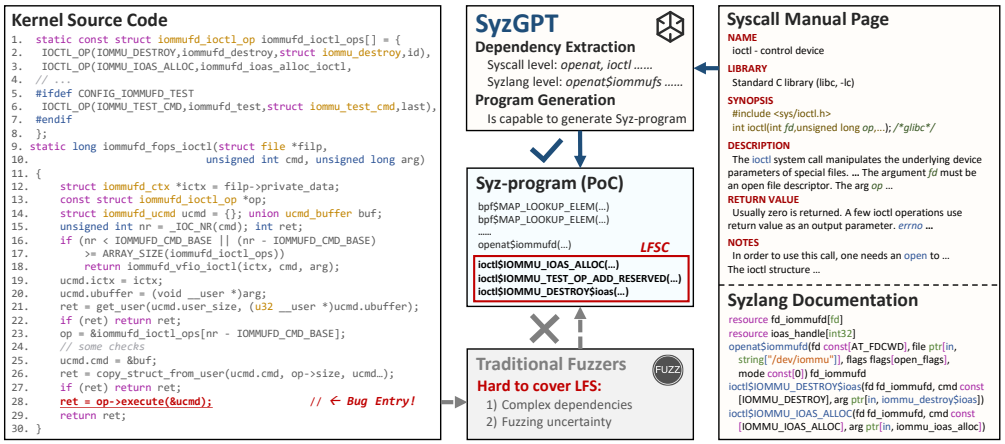


Fig. 1. A real-world example illustrating the limitation of LFSC and the motivation for our approach

With the rise of large language models (LLMs), massive amounts of data are being fed to LLMs, including documentation and code on the internet [17]. Given their capability in code comprehension and generation [16, 47, 58], we attempt to use LLMs to break through the limitations of generating high-quality programs for LFSC. According to the capability tests in the motivating example, we find that LLMs have the knowledge in the Linux kernel domain. Therefore, we can leverage LLMs to extract syscall dependencies and generate seeds for LFSC instead of generating them through an inefficient mutation process (addressing L1 and L2). However, there are still challenges when adopting LLMs to generate effective Syz-programs for kernel fuzzing.

**Challenges in using LLMs. C1:** *How to use LLM to generate valid seeds that meet the syntactic requirements for kernel fuzzing.* Although LLMs have shown excellent capabilities in security fields of software, deep learning library, and protocol [25, 26, 43, 65], it is impractical to directly utilize LLMs for Linux kernel fuzzing. The programs generated directly with simple instructions (*zero-shot* or *few-shot*) have meager syntactic validity (see Table 2). Programs with invalid syntax would be dropped by syntax checker and will not affect fuzzing. **C2:** *How to use LLM to generate quality seeds that improve the coverage of LFS.* Among the syntactically valid programs directly generated by LLM, many programs contain only one syscall without context or combinations of syscall sequences that are contextually ineffective. Using those programs with invalid context has little help with fuzzing.

**Our Approach.** To address the challenges and improve the coverage of LFS in kernel fuzzing, we propose SyzGPT, an LLM-assisted, automated kernel fuzzing framework for generating effective seeds for LFS with dependency-based retrieval augmented generation (DRAG). We design the following automated process to unleash the potential of LLM for kernel fuzzing. First, we use LLM to extract the syscall dependencies from Linux kernel manpage documentation to augment the dependencies extracted from Syzlang documentation through static analysis. Second, we provide LLM with program syntax knowledge to help it understand the Syz-program format (C1). Then, SyzGPT leverages the program knowledge in the runtime fuzzing corpus to enable LLM to infer the synopsis and dependencies of the target syscalls via retrieval augmented in-context learning (the term of few-shot learning [20]). Last, during the fuzzing loop, SyzGPT periodically extracts LFS from the current corpus as targets, constructs adaptive prompts for retrieval augmented in-context learning, generates and repairs effective seeds to enrich the fuzzing corpus (C2).

We implement SyzGPT based on Syzkaller (recent stable commit *f1b6b00*). On 436 sampled LFSC targets, SyzGPT achieves a seed generation valid rate of 87.84%, outperforming the state-of-the-art

LLM-based seed generator Fuzz4All [65] (16.74%). We also evaluate its fuzzing performance on three representative Long-Term-Support Linux kernels (6.6, 5.15, and 4.19), which are widely deployed in many distributions. Compared to state-of-the-art syscall-based fuzzers Syzkaller, MoonShine [45], Healer [56], ACTOR [28], MOCK [66], ECG [72], and KernelGPT [70], SyzGPT improves code coverage by 15.83%, 15.70%, 28.03%, 42.76%, 5.21%, 9.05%, and 7.52%, LFS coverage by 112.72%, 44.69%, 132.63%, 29.91%, 8.25%, 31.18%, and 46.60%, and vulnerability detection by 47.44%, 35.29%, 858.33%, 219.44%, 945.45%, 76.92%, and 79.69%, respectively. Besides, SyzGPT has independently discovered 26 unknown bugs (10 are LFSC-related), with 11 confirmed.

**Contributions.** We summarize our contributions as follows:

• **Novelty.** We propose the first automated LLM-assisted kernel fuzzing framework aiming at seed generation. We define and mitigate the LFS issue in kernel fuzzing. To address the challenges in generating effective seeds for LFS, we propose documentation-based syscall dependency augmentation (Section 3.2) and dependency-based RAG to unleash the capability of LLM (Section 3.3), and feedback-guided seed generation (Section 3.4) to enhance fuzzing performance.

• **Benchmark.** We introduce a benchmark for LLM-based kernel seed generation that includes a novel set of evaluation metrics and datasets. We also build a high-quality dataset for LLM fine-tuning (Section 4). To facilitate future comparisons and advancements in this domain, we have open-sourced [15] the implementation, benchmark, and fine-tuned LLM of SyzGPT.

• **Findings.** According to our evaluation, SyzGPT achieves improvements of 17.73% in code coverage, 58.00% in LFS coverage, and 323.22% in vulnerability detection across three LTS kernels. Notably, SyzGPT independently discovers 26 previously unknown real-world bugs with 10 LFS-related (Section 5). In addition, we derive several insights into applying LLMs to kernel fuzzing (Section 6).

## 2 Background and Related Work

### 2.1 Syscall-Based Kernel Fuzzing

The Linux kernel defines over 360 syscalls [6] for different module interfaces and exposes the kernel functionality to user-space processes. Therefore, the kernel fuzzers can test the kernel by invoking different syscall sequences, called syscall-based kernel fuzzers [34, 52]. Syzkaller is a state-of-the-art syscall-based kernel fuzzer, which utilizes Syzlang to define a pseudo-formal grammar for the precise specification of syscalls and provides a comprehensive way to model complex data structures and dependencies among syscalls. With the syscall specifications, Syzkaller could generate and mutate test cases effectively. However, ensuring the quality of Syzlang and fully supporting the various kernel modules and the growing drivers require expert knowledge and human efforts. Some studies try to tackle this problem by automated syscall description generation. KSG [55] finds syscall handler structures and the accessed data structure by probing the kernel dynamically. SyzDescribe [29] identifies common implementation patterns to generate specifications through static analysis. FuzzNG [21] proposes a simple alternative to Syzlang by API hooking, requiring configurations less than 1.7% the size of Syzlang's while achieving coverage comparable to Syzkaller. However, generating more syscall descriptions does not alleviate the problem of LFS, and may even introduce new LFS (e.g., we find 6 of these 12 newly defined syscalls [69] are LFS).

In addition, Syzkaller considers dependencies among syscalls by the mechanism of choice table, which records the priority that a syscall should be invoked after another syscall and guides the syscall selection during program generation and mutation. Many recent studies aim to improve fuzzing coverage by resolving syscall dependency. MoonShine [45] distills the dependencies from real-world programs execution traces. Healer [56] learns the relation table through dynamic execution. ACTOR [28] and StateFuzz [73] adopt different guiding strategies to model the behaviors of syscalls. SyzGen++ [23] extracts syscall dependencies and generates specifications through

symbolic execution and operation pairing. MOCK [66] uses a neural network language model to capture the context-aware dependencies of syscalls. However, the LFS can still be observed in fuzzing within a unit of time. As an intuitive solution, we can generate effective seeds for these LFS.

### 2.2 Seed Generation for Kernel Fuzzing

Seed generation is fundamental in the fuzzing domain, where providing higher-quality seeds enhances the fuzzing performance. However, unlike seeds generated for user-space fuzzing [49, 61], those used in kernel fuzzing are highly structured and format-sensitive. Syzkaller generates inputs with the guidance of the syscall description language and mutates inputs according to the choice table and hints. All inputs must satisfy the program syntax [9], or the checker will filter them out.

Due to the strict program syntax and complex syscall dependencies, it is hard to construct high-quality seeds manually. Although the Syzkaller project has provided several hundred expert-written default seeds in *sys/linux/test/*, these seeds can only cover 225 specialized calls (82 system calls), which leaves the efforts of synthesizing valid syscall sequences for LFSC to the fuzzing phase. However, as explained in Limitation L1, it is also inefficient to cover the LFSC by fuzzer mutation. Although some code oriented [54, 57] and vulnerability oriented [33, 76] approaches may impel fuzzers to explore such syscalls, the coverage of LFSC remains confined to the specific targets and the reliance of static analysis and dynamic execution introduce substantial performance overhead. Thus, direct generation of valid seeds for LFSC presents a superior solution. Unfortunately, our investigation reveals no existing work specifically addressing Syz-program generation. This critical technical gap establishes an important research challenge, motivating us to propose SyzGPT.

### 2.3 Large Language Model for Fuzzing

Large Language Models are Transformer-based [27, 36, 59] models pre-trained on massive textual datasets. Aligned with particular intentions [75], LLMs have advanced in many natural language processing tasks. Specifically, users can guide LLMs to handle specific tasks through prompts [44], which are framed as questions or instructions in natural language. Many approaches for designing prompts (*prompt engineering*) have been proposed to improve the performance of LLMs, such as Few-shot learning [20], Chain-of-thought [63] and Retrieval-augmented generation [37].

Recent studies have been exploring the potential of LLMs in code generation [39, 50, 62] and fuzzing. TitanFuzz [25] applies LLMs to generating test cases in deep learning library fuzzing. Fuzz4All [65] demonstrates that LLMs can serve as a universal fuzzer for various software systems. ProphetFuzz [60] leverages auto-prompting to predict and fuzz high-risk option combinations. ChatAFL [41] proves the effectiveness of LLMs for protocol fuzzing in enriching initial seeds and state inference. WhiteFox [68] utilizes LLMs to generate high-quality inputs in white-box compiler fuzzing. Unfortunately, none of these methods are applicable to the Linux kernel.

As a preliminary attempt to use LLMs in the domain of kernel fuzzing, a technical blog [18] and KernelGPT [70] show the feasibility of LLMs for syscall specification generation. Then ECG [72] further explores LLMs in generating C-based inputs for embedded OS kernel functions. However, none of them directly support seed generation in Syz-program format for kernel fuzzing or aim at alleviating LFSC problem. KernelGPT focuses on syscall specifications but not Syz-programs and may introduce new LFSC through its syscall definitions. ECG requires converting C programs to Syz-programs, a process proven inefficient due to its minimal conversion success rate (Section 5.1). Furthermore, since many LFSC-related kernel functions are inaccessible to user-space C programs without explicit syscall invocations, this indirect approach underperforms the direct syscall sequence generation via Syz-programs. Therefore, we propose SyzGPT, which leverages DRAG with historical knowledge to empower LLM-based Syz-program generation, ultimately enhancing kernel fuzzing coverage of both code and syscalls.

## 3 Design

In this section, we present the design of SyzGPT, an LLM-assisted kernel fuzzing framework for generating effective seeds for low frequency syscalls and improving the fuzzing performance. We first overview the framework and then describe how each component works.

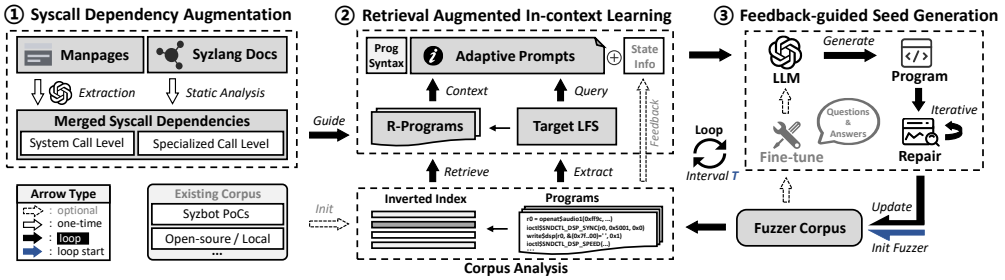


Fig. 2. Framework of SyzGPT

### 3.1 Overview

Figure 2 illustrates the framework of SyzGPT. In the preprocessing phase (white arrows), SyzGPT collects Linux manpage documentation and Syzlang specifications from the internet. It then extracts system call level dependencies from the manpages via LLM, augmenting the specialized call level dependencies extracted from Syzlang through argument-based static analysis (step ①). The merged dependencies guide the dependency-based RAG in step ②. In the loop phase (black arrows), SyzGPT maintains an inverted index of syscalls from the historical corpus (runtime fuzzer corpus and any existing corpus), serving as the program base for retrieving reference programs (R-programs[4]). It identifies LFS by comparing covered syscalls in the runtime corpus with the enabled syscall set, and retrieves $N$ R-programs for each target LFS based on the augmented dependencies. With program syntax and R-programs in context, SyzGPT performs retrieval-augmented in-context learning for target LFS (step ②). Next, SyzGPT adapts the seed generation strategy based on execution feedback from the runtime corpus and iteratively repairs invalid programs based on syntax feedback (step ③). Finally, it loads the generated seeds to enrich the corpus. With a preset update interval $T$, SyzGPT SyzGPT periodically repeats steps ②-③ to generate new seeds for the current LFS, while automatically building a dataset for fine-tuning if needed.

### 3.2 Syscall Dependency Augmentation

According to the mechanism of choice table, Syzkaller-like fuzzers primarily learn the dynamic syscall dependencies from correlations observed in the corpus. However, acquiring these dependencies requires significant fuzzing time to cover as many syscall combinations as possible, and some LFSC may still be missing. To address this, we propose leveraging LLMs to extract syscall dependencies from kernel documentation (e.g., manpages) before fuzzing (① in Figure 2), as these documents contain rich descriptions of both explicit and implicit syscall dependencies.

Figure 3 shows the workflow of extracting syscall dependency from Linux manpage using LLM. First, SyzGPT determines whether to condense the manpage documentation by comparing the total length of all sections against the LLM's context limit. To encourage comprehensive extraction, we provide definitions and examples of explicit and implicit syscall dependencies in the first round of extraction. We define explicit dependencies as relationships where one syscall directly relies on

[4]Throughout this paper, we refer to R-programs as programs containing syscalls on which the target syscall depends.

the result or output of another. For example, syscall `accept` explicitly depends on `select`, `poll`, `epoll` as the manpage states: *"you can use select, poll, or epoll. A readable event will be delivered, and you may then call accept to get a socket for that connection"*. Implicit dependencies refer to indirect relationships that arise from the kernel's overall behavior or state management. For example, `futex` implicitly depends on `mmap` or `shmat` for shared memory operations, as stated in the manpage: *"In order to share a futex between processes, the futex is placed in a region of shared memory, created using (for example) mmap or shmat"*. In the second round of dependency extraction, the LLM is prompted to complete the dependency set based on its heuristic knowledge. Finally, the results of both rounds are merged to form the system call level dependency set.

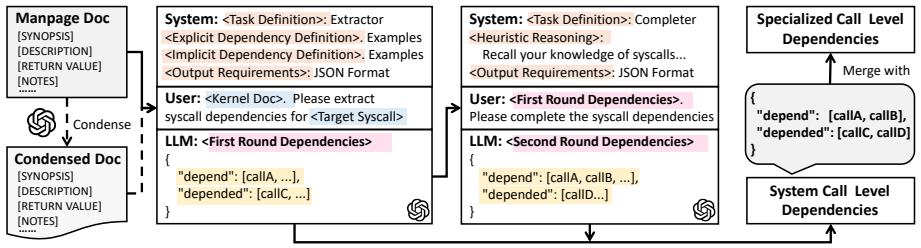


Fig. 3. Workflow of syscall dependency extraction and augmentation

We extract the specialized call level dependencies from Syzlang by analyzing the *in*, *out*, and *inout* (means that the argument can be both input and output) directions of syscall arguments and returns. If a syscall consumes an argument with direction *in* or *inout*, it should be invoked after a syscall which produces the corresponding resource through return (direction *out*) or argument (direction *inout*). For example, `ioctl$ASHMEM_GET_PROT_MASK` has three arguments *fd (fd_ashmem)*, *cmd (const[30470, const])*, and *arg (ptr[out, ashmem_pin])*. And the return argument of `openat$ashmem` is *fd_ashmem*, so a dependency is built through *fd_ashmem*. Moreover, `ioctl$ASHMEM_SET_PROT_MASK` has an argument *arg (ptr[in, ashmem_pin])*, which could be matched with the *out* argument of `ioctl$ASHMEM_GET_PROT_MASK`. Finally, two levels of dependencies are synthesized as the augmented syscall dependencies, which will guide the R-programs retrieval in step ②. Then, SyzGPT steps into the loop phase (Figure 2 steps ②-③) for seed generation.

### 3.3 Retrieval-Augmented In-Context Learning

According to our observations, though using the few-shot approach to generate Syz-programs slightly increases the valid program rate from 11.47% to 26.84% and average program length from 1.24 to 2.80 compared to zero-shot generation, the overall performance remains poor. This modest improvement occurs because the context can serve as the format demonstration to LLM. However, the fixed context results in high content coupling between the generation and fixed programs, leading to many erroneous syscall invocations and combinations. These issues highlight the need for a more adaptive approach. The Retrieval-Augmented Generation (RAG) is fitting for dynamically adjusting context based on varying targets. However, retrieving suitable context for the LFS remains a significant challenge. Therefore, we design a dependency-based retrieval-augmented in-context learning approach to construct adaptive prompts for different LFS to ensure syntactic and contextual effectiveness of program generation, corresponding to ② in Figure 2.

**Corpus Analysis for LFS.** Before constructing the adaptive prompts, generation targets must be determined. SyzGPT automatically calculates the current LFS by comparing the syscalls covered in the corpus with the enabled syscall set (usable syscalls determined at the beginning of fuzzing,

introduced in Section 1). Meanwhile, SyzGPT establishes an inverted index [64, 74] for every syscall by traversing the historical corpus, including the runtime corpus and the optional existing corpus (e.g., Syzbot PoCs [5] and MoonShine corpus). The inverted index maps syscalls to the file hashes of the programs containing the syscall, which speeds up the following R-programs retrieval procedure.

**Dependency-based R-programs Retrieval.** Since blindly retrieving programs from the historical corpus would add many completely irrelevant syscalls to the context, we filter out the R-programs that are valuable to the target syscall based on their dependencies. These R-programs can provide adequate syntax information, parameter value references, and sequential inspiration for generating programs for target syscalls. We define the syscalls that appear before the target syscall as *preceding syscalls* and their *preceding syscalls* as *multi-hop preceding syscalls*. Because each syscall that the target syscall depends on must belong to its *preceding calls*, we design a dependency-based R-programs retrieval algorithm to evenly select syscalls across different dependency levels.

**Algorithm 1:** Dependency-based R-programs Retrieval

**Data:** Historical corpus $P_h$, default seeds $P_d$, Augmented syscall dependencies $D_a$
**Input:** Target specialized call $c_t$, few-shot number $N$
**Output:** Retrieved R-programs $P$

1 $P \leftarrow set()$, $N_{hop} \leftarrow N$, $cnt \leftarrow 0$ ; // variable initialization
2 $s_t \leftarrow$ SplitCall($c_t$) ; // e.g., mmap←mmap$xdp
3 **for** $k \leftarrow 1$ **to** $N_{hop}$ **do**
4 $c[k], s[k] \leftarrow$ GetDependence($D_a, c[k-1], s[k-1]$) ; // $c[0] = c_t$, $s[0] = s_t$
5 **for** $k \leftarrow 1$ **to** $N_{hop}$ **do**
6 $c_k \leftarrow$ RandomChoose($c[k]$)
7 **if** $P \leftarrow P \cup SearchProgram(P_h, c_k)$ **then**
8 $cnt \leftarrow cnt + 1$
9 **break if** $cnt \geq N$
10 $s_k \leftarrow$ RandomChoose($s[k]$)
11 $c'_k \leftarrow$ ChooseCall($s_k$)
12 **if** $P \leftarrow P \cup SearchProgram(P_h, c'_k)$ **then**
13 $cnt \leftarrow cnt + 1$
14 **break if** $cnt \geq N$
15 **for** $i \leftarrow len(P) + 1$ **to** $N$ **do**
16 $P \leftarrow P \cup$ ChooseProgram($P_d$)
17 **return** $P[0:N]$

Algorithm 1 shows the R-programs retrieval process, SyzGPT takes historical corpus $P_h$, Syzkaller default seeds $P_d$, augmented syscall dependencies $D_a$ as known data. It receives target specialized call $c_t$ (e.g., LFSC) and the shot number $N$ as input and outputs the retrieved R-programs $P$. Firstly, SyzGPT initializes dependencies query hop $N_{hop}$ with $N$ (Lines 1), representing that it will search up to $N$ hops (e.g., `munmap` depends on `mmap`, `mmap` depends on `brk`, and `brk` is the 2-hop *preceding syscall* of `munmap`) of dependencies. SyzGPT splits target specialized call $c_t$ into syscall $s_t$ to take syscall-level dependencies into account, and then queries the $N_{hop}$ dependence of $c_t$ and $s_t$ from the augmented syscall dependencies $D_a$, which will be stored in $c[k]$ and $s[k]$ respectively (Lines 2-4). Secondly, SyzGPT randomly chooses a specialized call $c_k$ from the k-hop dependencies $c[k]$ and tries to retrieve one program containing $c_k$ from historical corpus $P_h$ and add it to $P$ (Lines 6-9). Function *SearchProgram* glances the corpus and retrieves one program that contains $c_k$ and has moderate program length. As for $s_k$, SyzGPT randomly chooses one of its specialized calls $c'_k$ from the enabled specialized call set and performs the same procedure on $c'_k$ (Lines 10-14). Thirdly, SyzGPT fills $P$ with programs from the default seeds $P_d$ up to the maximum length $N$ (Lines 15-17). Finally, it returns the retrieved R-programs set $P$ ($N$ is usually smaller than 5 in practice [22]).

**Prompt Construction for In-Context Learning.** Unlike common programming languages where few-shot with simple instructions often yields effective program generation, Syz-programs follow a niche format defined exclusively by the Syzkaller project. This specialization is one of the reasons why our zero-shot and few-shot Syz-program generation tests yield poor results. To address this, we incorporate syntax learning and retrieval-augmented in-context learning to enhance LLM's ability to generate Syz-programs.

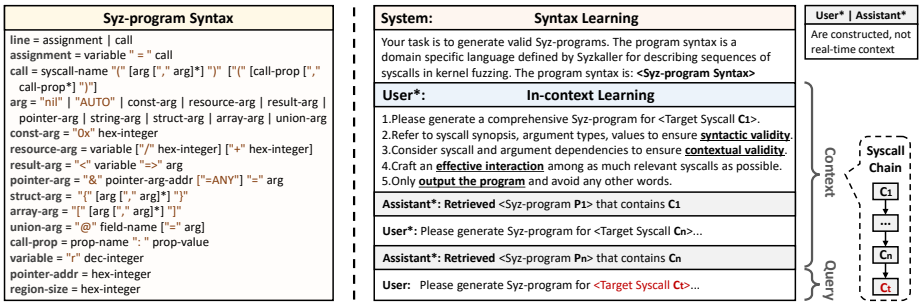


Fig. 4. Adaptive prompts of retrieval-augmented in-context learning for LFS $c_t$

Figure 4 describes the adaptive prompts used by SyzGPT for program generation. In the system prompt, we assign the task of generating a comprehensive Syz-program to the LLM and teach it the Syz-program syntax based on the domain-specific language definition (on the left side of Figure 4). When constructing adaptive prompts, SyzGPT automatically creates $N$ pairs of conversation history between *User* and *Assistant*, where retrieved R-programs are provided as context. Specifically, the user query consists of a request to generate a program containing the target syscall $c_i$ ($i \in [1, N]$), along with emphases on syntactic validity, contextual validity, effective interaction, and output format. Notably, $c_1$ to $c_n$ form a syscall chain to target $c_t$, where $c_i$ is the *preceding syscall* of the next syscall. The assistant's response is the retrieved program $p_i$ that contains $c_i$.

### 3.4 Feedback-Guided Seed Generation for Fuzzing

With adaptive prompts for the target $c_t$, SyzGPT can generate Syz-program accordingly. However, due to strict syntax checks, some programs may still exhibit invalid syntax caused by minor syntax errors or invalid context of unreasonable call sequences. Therefore, we propose program repair with syntax feedback and program re-generation with execution feedback to address these issues correspondingly. The workflow of feedback-guided seed generation is shown in Figure 5. We also provide a running example in Figure 6 at the end of the section for better understanding.

Table 1. Heuristic repair operations to address different types of syntactic illegality

| Syntax Error Type | Repair Operation |
|---|---|
| ❶ Unknown syscall *SYSCALL* | Calculate the cosine similarity between *SYSCALL* and each specialized call that shares the same base system call, and replace *SYSCALL* with the five most similar specialized calls. |
| ❷ Want *STR_A*, got *STR_B* | Utilize regular expression to replace the failure character *B* with *A*. |
| ❸ Unexpected *EOF* | Fixe the last syscall by multi-level parentheses completion and position arguments filling, since most EOFs occur at the last syscall invocation due to the LLMs Parroting Problem [19]. |
| ❹ Escaping filename *FILE* | Replace the illegal filenames with preset strings, such as *"/dev/kvm"* and *"/tmp/file0"*. |
| ❺ Out of *MaxCalls* | Trim lines exceeding the limit only when the target syscall has already included in previous lines. |

**Program Repair with Syntax Feedback.** Since Syzkaller's syntax checker only reports the first error it encounters, repairing all potential syntax errors via LLM is impractical, as it would require multiple rounds of interaction and additional code snippets as context, leading to significant time and token overhead. Therefore, we define several heuristic repair operations based on our empirical observations to handle five main types of syntactic errors, as shown in Table 1. First, SyzGPT reads the program into *lines* and fixes common faults by replacing double quotes with single quotes, removing trailing semicolons, and converting unprintable *ascii* characters to hexadecimal representation. Then, SyzGPT repairs the program iteratively, where it analyzes the syntax error *err* reported by the checker and applies a corresponding repair operation. SyzGPT regards each repair operation as one repair attempt and breaks the loop when the number of attempts exceeds $R_{try}$ or *err* is *None*. Finally, SyzGPT returns the repaired program.

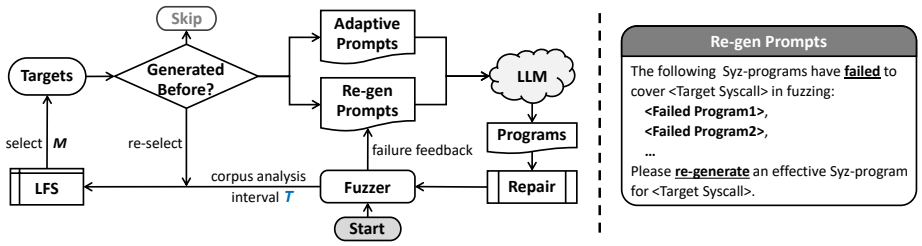


Fig. 5. Workflow and prompt details of feedback-guided seed generation

**Program Re-generation with Execution Feedback.** According to Figure 5, in each generation round, SyzGPT analyzes LFS from the corpus and randomly selects $M$ LFS as generation targets. For each target, SyzGPT adopts different generation strategies. If a target syscall has never been selected, SyzGPT retrieves R-programs and constructs the adaptive prompts. If the target has failed fewer times than a preset generation *limit*, SyzGPT attempts re-generation or re-selects another LFS with a *probability*. If the failures exceed the *limit*, SyzGPT skips the target. The prompts for re-generation include the program syntax in the system prompt and the context containing previously generated programs that failed to cover the target syscall, as shown at the right of Figure 5.

**Fuzzing Loop and Fine-tuning Support.** In the fuzzing loop, the seed generation process is triggered after each interval $T$. After generating and repairing seeds for a batch of current target syscalls, SyzGPT updates the current runtime corpus with the contextually valid programs and waits for the next round of operations. Meanwhile, SyzGPT backs up the pairs of query prompts and contextually valid programs during the fuzzing loop, which can serve as the datasets for instruction fine-tuning of offline LLMs under the paradigm of "*question*" and "*answer*".

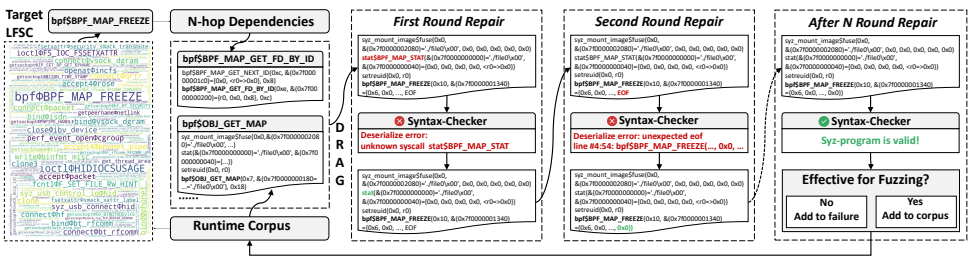


Fig. 6. A running example for intuitive understanding the seed generation of SyzGPT

## 4 Implementation

In this section, we introduce the implementation of SyzGPT (8k lines of Python and Go), including dataset collection, dependency extraction, seed generation, fuzzer integration, and LLM fine-tuning.

**Dependency Extraction and Augmentation.** We crawl the documentation of 271 Linux system call from the Linux Manual Page [7] and collect 187 Syzlang files defining the synopsis and resources of 4446 specialized calls in Syzkaller. Then, we implement the system call level dependency extractor based on OpenAI API and the specialized call level dependency extractor based on Python. To ensure the determinism of syscall dependency extraction, we set the temperature of LLM to 0.3. Due to the lack of dependency ground truth, we evaluate the syscall dependency extraction effectiveness through manual analysis (81.37% dependencies of 15 sampled syscalls are correct).

**Dependency-based RAG.** To analyze LFS, we extend Syzkaller by modifying the RPC structure in *rpctype.Input* to record covered syscalls. We implement dependency-based R-programs retrieval algorithm by an inverted index mapping syscalls to related filenames and regex-based searching. We implement *SyzGPT-generator* based on OpenAI API, with a temperature of 0.7 to introduce more variability in program generation. In particular, *SyzGPT-generator* generates seeds for 100 selected targets every 1 hour, where it builds adaptive prompts or constructs re-generation prompts with a failure *limit* of 3 and a *probability* of 0.1. To repair the generated programs, we implement *syz-repair* based on Syzkaller, where $R_{try}$ is 25 as a balance of repairing accuracy and efficiency. Notably, we do not explore the optimal parameter values, as they are already satisfying.

**Fuzzer Integration and LLM Serving.** Based on Syzkaller, we implement the seeds loading by adding a *Go routine* in *syz-manager*. We specify the generation interval $T$ and the directory where seeds will be generated and repaired at the beginning of running *SyzGPT-fuzzer*. Additional modules for experiments are also added to track syscall coverage, rawcover and corpus statistics. In terms of LLM fine-tuning, we implement the parameter efficient fine tuning (PEFT) with PyTorch based on LoRA (Low-rank Adaptation [31]) for CodeLlama-7b-Instruct-hf. We collect the dataset (8k train and 3k validation) from the context-valid programs generated by GPT-3.5-turbo-16k-0613 during our fuzzing experiments, and obtain CodeLlama-syz by tuning on 4 × A800 GPUs with *batch_size* 2 for 2 *epochs* in 16 hours. We adopt *fastchat* with *vllm* to deploy all the offline and fine-tuned LLMs.

## 5 Evaluation

In this section, we evaluate SyzGPT to answer the following key research questions:

- **RQ1. Seed Generation.** What is the seed generation performance of SyzGPT?
- **RQ2. Fuzzing.** What is the fuzzing performance of SyzGPT compared to state-of-the-art tools?
- **RQ3. Ablation study.** How does each component contribute to the performance of SyzGPT?
- **RQ4. Real-world Bug Discovery.** Can SyzGPT discover real-world and LFSC-related bugs?

**Platform.** Our experiments are conducted on an Ubuntu 20.04 server with Intel Xeon CPU E7-4850 v4 @ 2.8GHz 128 cores and 256GB memory. The LLMs fine-tuning and serving are done on an Ubuntu 22.04 server with Intel Xeon Gold 5218R @ 4.0GHz 512GB memory, and 2×A800 GPUs.

**Experimental Setup.** For seed generation experiments, we use a 24-hour corpus generated by default Syzkaller as the existing corpus, since no runtime fuzzer corpus is available for SyzGPT. The versions of the base LLMs are: GPT-3.5-turbo-16k-0613, GPT-4-0613, Claude-3-5-sonnet-0620, CodeLlama-7b-Instruct, Vicuna-7b-1.5-16k, Llama-3-8b-Instruct. For fuzzing experiments, we compile three target LTS kernels (6.6.12, 5.15.140, 4.19.300) with the Syzbot configs [11] accordingly. We allocate 8 QEMU virtual machines (2vCPUs and 4GB memory) for every fuzzer. We continuously fuzz the target kernels for 24 hours (192 CPU hours) and repeat the experiments five times (by excluding the best and worst groups from seven runs) to avoid the effects of uncertainty. To ensure fair comparison, all fuzzers are initialized with the same default seeds from Syzkaller project.

**Evaluation Metrics.** As an LLM-based seed generation method, we evaluate the syntactic effectiveness, semantic effectiveness, and costs of the generation of SyzGPT under different LLMs. As a fuzzing framework for Linux kernel, we evaluate the performance of SyzGPT in overall code and syscall coverage. We define the metrics as follows:

• *Syntax Valid Rate (SVR):* We define the programs that pass Syzkaller syntax checks as syntactically valid. We evaluate the seed generation performance in syntactic validity by SVR ($\frac{N_{syn}}{N_{total}}$), where $N_{syn}$ and $N_{total}$ are the numbers of syntax valid programs and total generated programs. $N'_{syn}$ is the number of syntax valid programs with length greater than 1. The higher the SVR and $N'_{syn}$, the better SyzGPT performs in seed generation.

• *Context Effective Rate (CER):* We use CER to evaluate the seed generation performance in terms of contextual effectiveness. The contextual effectiveness of a program represents the diversity of the syscall sequences and the capability to cover more code branches. We define a program as context effective if its execution coverage exceeds the sum of its individual syscalls' coverage. Specifically, we split a program $P$ (i.e., $P = \{c_1, c_2, ..., c_n\}$) into $n$ programs $P_{c_i}$ (only contains one syscall $c_i$). The contextual effectiveness of $P$ can be judged as: $bool(Cov(P) > \sum_{i=n}^{i=1} Cov(P_{c_i}))$. Then, we compute CER=$\frac{N_{con}}{N_{syn}}$, where $N_{con}$ is the number of context effective programs. The higher the CER, the higher the quality of the generated seeds, which are more likely to contribute to fuzzing.

• *Program Diversity:* We leverage average program length ($\overline{L}$=$\frac{\sum L_i}{N_{syn}}$) and average syscall number ($\overline{N_s}$=$\frac{N_{syn}}{N_{syn}}$) to represent the generation diversity, where $L_i$ and $N_s^i$ are the sequence length and syscall number of the $i$-th program with valid syntax. The higher the $\overline{L}$ and $\overline{N_s}$, the more diverse the program generation, which is another manifestation of contextual effectiveness.

• *Code and Syscall Coverage:* Code coverage is a built-in metric of every coverage-guided kernel fuzzer with kcov at the basic block level. We define unique coverage ($Cov_u(A|B) = |\, Cov(A) - Cov(A) \cap Cov(B)\,|$) as the number of unique basic blocks of fuzzer $A$ that $B$ cannot cover, then we use unique coverage rate ($R_u(A|B)$=$\frac{Cov_u(A|B)}{Cov_u(B|A)}$) to estimate $A$'s ability to find unique coverage compared to $B$. We define rate of covered syscalls ($R_s$=$\frac{N_s}{N_{enabled}}$), where $N_s$ and $N_{enabled}$ are numbers of covered and enabled syscalls. The higher the coverage metrics, the better the fuzzing performance.

• *Valuable Input Number ($N_{vi}$):* We define a fuzzing input valuable if it aligns with the fuzzer's guiding strategy, such as triggering new coverage or signal. We use $N_{vi}$ to represent the number of valuable inputs during fuzzing, which can be captured by the *bench* mechanism of Syzkaller. The higher the $N_{vi}$, the better the fuzzing performance in code coverage metrics.

### 5.1 RQ1: Seed Generation Performance

We evaluate the seed generation performance of SyzGPT in terms of syntactic validity, contextual effectiveness and generation cost. As shown in Table 2, we choose seven different base LLMs: GPT-3.5, GPT-4 and Claude-3.5-sonnet [46] represent online chat LLMs, Vicuna [24] and Llama-3 [42] represent offline chat LLMs, CodeLlama stands for open-source LLMs with code knowledge, and CodeLlama-syz is our fine-tuned version of CodeLlama based on context effective programs. We extract LFSC from the scope of five 24-hour Syzkaller runs and sample 436 LFSC (approximately 25%) as generation targets, ensuring a diverse distribution across system calls. We compare our method against zero-shot, few-shot, Fuzz4All and ECG methods under different LLMs. Zero-shot adopts the User prompts in Figure 4, representing the capability of LLM itself. Few-shot consists of 3 fixed queries and samples. Fuzz4All and ECG use their default prompts. KernelGPT is not included in this experiment, as it is not designed for generating seeds. The results are shown in Table 2, where our method demonstrates effectiveness and superiority.

Table 2. Performance of program generation methods on different LLMs

| Method | LLM | Syntactic Validity | | | Contextual Effectiveness | | | | OER | Generation Cost | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | $N_{syn}$ | SVR | $N'_{syn}$ | $N_{con}$ | CER | $\overline{L}$ | $\overline{N_s}$ | | Tokens | Time (s) |
| SyzGPT | GPT-3.5 | **383** | **87.84%** | 285 | 186 | 48.56% | 3.91 | 3.12 | 42.66% | 1769.33 | 4.66 |
| | GPT-4 | 311 | 72.48% | 310 | **240** | **77.17%** | **6.46** | **5.11** | **55.05%** | 1496.04 | 12.04 |
| | Claude-3.5-Sonnet | 319 | 73.17% | 318 | **261** | 81.82% | **10.83** | **8.37** | **59.86%** | 1856.10 | 6.69 |
| | CodeLlama | 358 | 82.11% | **352** | 147 | 41.06% | 4.82 | 3.13 | 33.71% | 2279.84 | 20.10 |
| | CodeLlama-syz | **383** | **87.84%** | 323 | 205 | 53.52% | 3.66 | 2.41 | 47.01% | 2422.68 | 8.55 |
| | Llama-3 | 300 | 68.81% | 287 | 190 | 63.33% | 5.73 | 4.55 | 43.58% | 2036.24 | 4.38 |
| | Vicuna-v1.5 | 213 | 48.85% | 27 | 12 | 5.63% | 1.22 | 1.15 | 2.75% | 1658.63 | 1.05 |
| Fuzz4All | GPT-3.5 | 73 | 16.74% | 57 | 53 | 72.60% | 5.41 | 3.59 | 12.16% | 830.80 | 7.20 |
| ECG | GPT-3.5 | 57 | 13.07% | 57 | 55 | **96.49%** | 4.95 | 3.14 | 12.61% | 535.98 | 7.23 |
| Few-shot | GPT-3.5 | 117 | 26.83% | 66 | 34 | 29.06% | 2.80 | 2.61 | 7.80% | 1887.86 | 5.50 |
| Zero-shot | GPT-3.5 | 50 | 11.49% | 6 | 3 | 6.00% | 1.43 | 1.10 | 0.70% | 502.15 | 12.94 |
| | CodeLlama | 0 | 0% | 0 | 0 | 0% | 0 | 0 | 0% | 1114.24 | 11.49 |
| | CodeLlama-syz | 212 | 48.62% | 130 | 93 | 43.87% | 2.57 | 1.78 | 21.33% | 925.24 | 7.27 |

ECG: We modified ECG through prompts migration, as it does not inherently support the seed generation task with syscalls as target.

**Syntactic Validity.** To calculate the number of generated programs with valid syntax ($N_{syn}$), we implement a syntax validator based on Syzkaller's program deserialization [8]: A program is considered syntactically valid if it can be successfully deserialized from *bytes* into a structured *prog*.

According to Table 2, by comparing SVRs over different methods, we find it hard for LLMs to understand the program syntax in zero-shot, few-shot, and Fuzz4All, which can be addressed by our method. The superiority of SyzGPT indicates that our prompting method significantly improves the syntactic validity of seed generation. ECG also underperforms due to two inherent flaws: (1) Generated C programs lack kernel-space function access, limiting syscall scope. (2) The conversion pipeline (C→trace→Syz-program) suffers cascading failures: only 141/436 C programs compile successfully, 55 of which even fail during execution with incomplete traces due to parameters/environment dependencies; despite our patches enabling 137/141 *trace2syz* conversions (originally 0%), only 57 Syz-programs contain target syscalls (SVR=13.07%). By comparing SVRs of different LLMs, we find that LLMs with larger scale generally bring higher SVRs (GPT-4 has lower SVR but higher $N'_{syn}$ than GPT-3.5 because it focuses on longer but error-prone programs). The code related knowledge, especially Syz-program related knowledge, can help LLMs to generate seeds with higher SVR. Using fine-tuned CodeLlama-syz can improve SVR of Zero-shot from 0% to 48.62% and SVR of SyzGPT from 82.11% to 87.84%. In addition, the proportion of $N'_{syn}$ to $N_{syn}$ (GPT-4: 310/311, Claude-3.5-Sonnet: 318/319, Llama-3: 287/300) indicates that increasing model complexity can also significantly improve the quality of generated programs, which will be further demonstrated in metrics of program diversity.

**Contextual Effectiveness.** To calculate CER, we obtain the overall program coverage and individual syscall coverage by Syzkaller's *syz-execprog* [10]. Note that we do not include programs with a length of 1 in the measurement of $N_{con}$, as they have no context. And we define overall effectiveness rate (OER=$N_{con}/N_{total}$) to represent the overall ability of a method to generate effective seeds.

The results in Table 2 highlight the effectiveness and diversity of the generated programs across different models. By comparing methods using the same LLM, SyzGPT achieves the highest OER (42.66% with GPT-3.5), validating its effectiveness. Fuzz4All and ECG exhibit higher CER due to limited $N_{syn}$ and ECG's C2Syz conversion (Compilable C programs are likely to be context effective). We also find fine-tuning CodeLlama with context effective programs can boost the generation performance of SyzGPT and Zero-shot. By comparing SyzGPT across LLMs, we find using Claude-3.5-Sonnet achieves the highest $N_{con}$ (261), OER (59.86%), $\overline{L}$ (10.83) and $\overline{N_s}$ (8.37), followed by GPT-4, CodeLlama-syz and Llama-3, while the smaller chat-focuesd Vicuna-v1.5 lags significantly. This demonstrates that the advancements in reasoning and specialized code knowledge can enable

SyzGPT to generate longer and more effective sequences. To present the program diversity more intuitively, we display the distribution of $\overline{L}$ in Figure 7, which also indicates that the more code knowledge and stronger reasoning ability the model has, the better the generation diversity.

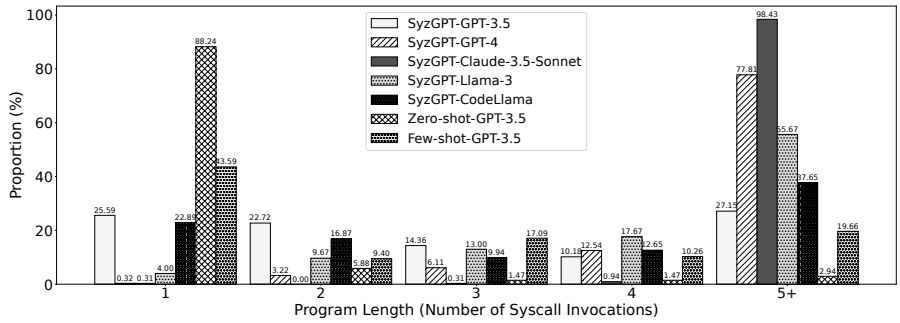


Fig. 7. Distribution of the lengths of syntax valid programs generated by different methods and models

**Generation Cost.** In our method's default configuration (GPT-3.5), a high-quality program can be generated within a few seconds and fewer than 2k tokens (0.007$), which is quite efficient and acceptable. For the cost of program repair, it takes an average of 5 milliseconds per program, which is accurate and fast enough, eliminating the need to use LLM for repair. For the computational costs, the online LLMs incur no cost, while the offline LLMs require approximately 14GB GPU memory for inference, 56GB GPU memory and 16 GPU hours for fine-tuning. For the cost of historical corpus, since no runtime fuzzer corpus is available in this experiment, we use the corpus from a default Syzkaller instance as historical corpus, which requires 24 hours to generate.

Table 3. Average results of fuzzing within 24 hours. The table shows the coverage ($Cov$), unique coverage/unique coverage rate ($Cov_u/R_u$), unique coverage of LFSC ($Cov_u^l$), number of valuable inputs ($N_{vi}$), number/rate of covered specialized calls ($N_s/R_s$), number of covered LFSC ($N_l$), and improvement (IMP) of SyzGPT compared with others, respectively.

| Version | Fuzzer | $Cov$ | ←IMP | $Cov_u$ / $R_u$ | ←IMP | $Cov_u^l$ | ←IMP | $N_{vi}$ | ←IMP | $N_s$ / $R_s$+ | ←IMP | $N_l$ | ←IMP |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 6.6 | Syzkaller | 163370 | 13.06% | *baseline* | 323.00% | 7110 | 16.67% | 29084 | 26.77% | 2693 / 69.09% | 13.00% | 278 | 111.51% |
| | MoonShine | 161526 | 14.35% | 19530 / 0.94 | 343.62% | 6935 | 19.61% | 30362 | 21.44% | 2682 / 68.80% | 13.46% | 414 | 42.03% |
| | Healer | 148585 | 24.31% | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ | 2166 / 55.57% | 40.49% | 327 | 79.82% |
| | ACTOR | 125166 | 47.57% | ✖ | ✖ | ✖ | ✖ | 24951 | 47.77% | 2543 / 65.24% | 19.66% | 443 | 32.73% |
| | MOCK | 173075 | 6.72% | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ | 2792 / 71.63% | 8.99% | 540 | 8.89% |
| | ECG | 164514 | 12.27% | 20731 / 0.99 | 326.26% | 6919 | 19.89% | 33364 | 10.51% | 2748 / 70.50% | 10.74% | 429 | 37.06% |
| | KernelGPT | 173342 | 6.56% | 26203 / 1.98 | 163.13% | 7584 | 9.38% | 34463 | 6.99% | **3241** / 63.25% | 23.43% | 408 | 44.12% |
| | **SyzGPT** | **184705** | - | **33960 / 3.23** | - | **8295** | - | **36871** | - | 3043 / **78.07%** | - | **588** | - |
| 5.15 | Syzkaller | 161032 | 20.32% | *baseline* | 643.00% | 7999 | 25.03% | 29646 | 33.32% | 2658 / 68.40% | 15.80% | 302 | 113.58% |
| | MoonShine | 163192 | 18.73% | 23083 / 1.22 | 527.05% | 7670 | 30.39% | 27494 | 43.76% | 2596 / 66.80% | 18.57% | 425 | 51.76% |
| | Healer | 149201 | 29.86% | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ | 1902 / 48.94% | 61.83% | 253 | 154.94% |
| | ACTOR | 144220 | 34.35% | ✖ | ✖ | ✖ | ✖ | 33294 | 18.72% | 2761 / 71.05% | 11.48% | 518 | 24.52% |
| | MOCK | 179810 | 7.76% | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ | 2885 / 74.24% | 6.69% | 600 | 7.50% |
| | ECG | 178767 | 8.39% | 34271 / 3.70 | 173.78% | 8623 | 15.98% | 32525 | 21.52% | 2741 / 70.54% | 12.29% | 447 | 44.30% |
| | KernelGPT | 183232 | 5.74% | 37528 / 3.40 | 189.12% | 8711 | 14.81% | 34059 | 16.05% | **3290** / 63.96% | 23.84% | 455 | 41.76% |
| | **SyzGPT** | **193758** | - | **44686 / 6.43** | - | **10001** | - | **39525** | - | 3078 / **79.21%** | - | **645** | - |
| 4.19 | Syzkaller | 152885 | 14.10% | *baseline* | 317.00% | 4173 | 17.35% | 32498 | 48.70% | 2764 / 74.88% | 9.44% | 268 | 113.06% |
| | MoonShine | 152988 | 14.02% | 18269 / 1.03 | 307.77% | 4124 | 18.74% | 33655 | 43.59% | 2728 / 73.91% | 10.89% | 407 | 40.29% |
| | Healer | 134257 | 29.93% | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ | 1772 / 48.01% | 70.71% | 217 | 163.13% |
| | ACTOR | 119178 | 46.37% | ✖ | ✖ | ✖ | ✖ | 29242 | 65.26% | 2621 / 71.01% | 15.41% | 431 | 32.48% |
| | MOCK | 172449 | 1.15% | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ | 2799 / 75.83% | 8.07% | 527 | 8.35% |
| | ECG | 163815 | 6.48% | 26173 / 1.91 | 165.97% | 4473 | 9.48% | 43652 | 10.70% | 2877 / 77.95% | 5.14% | 509 | 12.18% |
| | KernelGPT | 158193 | 10.27% | 22286 / 1.49 | 212.75% | 4237 | 15.58% | 35424 | 36.42% | **3146** / 65.34% | 25.44% | 371 | 53.91% |
| | **SyzGPT** | **174435** | - | **32547 / 3.17** | - | **4897** | - | **48324** | - | 3025 / **81.96%** | - | **571** | - |

✕: Healer and MOCK do not support dumping *rawcover* and *valuable inputs* as they are not Syzkaller-based fuzzers.
✖: ACTOR does not support comparing unique coverage as the address map of its patched kernel is different.
+: Default $N_{enabled}$ of three kernels are 3898, 3886 and 3691, while KernelGPT enables 5124, 5144 and 4815. And we do not stat the LFSC newly introduced by KernelGPT for $N_l$.

### 5.2 RQ2: Fuzzing Performance

To evaluate SyzGPT's fuzzing performance, we compare it with state-of-the-art syscall-based kernel fuzzers (Syzkaller, MoonShine, Healer, ACTOR, MOCK, ECG, and KernelGPT) in terms of coverage and vulnerability detection. Among these fuzzers, MoonShine and ECG focus on optimizing the initial corpus, while Healer and MOCK aim to improve syscall relationship learning. ACTOR adopts an action-guided strategy instead of coverage-guided, and KernelGPT focuses on optimizing syscall specifications. All fuzzers use default seeds from *syzkaller/sys/linux/test/*. Additionally, MoonShine uses *strong_distill.db*, ECG takes the LLM-generated programs for all LFSC as initial *corpus.db*, and KernelGPT employs the LLM-generated syscall specifications which define more specialized calls. The experiments are conducted on three LTS kernels as detailed in Setup.

**Code and Syscall Coverage.** Table 3 shows the average fuzzing results of different fuzzers on three kernels for five repeats of 24 hours in code and syscall coverage metrics. We compute $Cov_u^l$ by attributing the covered addresses to the functions that can only be reached from the LFSC syscall entry in the kernel call graph [40], reflecting the fuzzer's ability to cover unique code related to LFSC. Addresses indirectly covered by LFSC are not included, as they are hard to track.

According to Table 3, SyzGPT outperforms all state-of-the-art fuzzers across every metric. Compared to Syzkaller, MoonShine, Healer, ACTOR, MOCK, ECG, and KernelGPT, SyzGPT achieves 15.83%, 15.70%, 28.03%, 42.76%, 5.21%, 9.05%, and 7.52% average coverage improvements. Specifically, SyzGPT shows significant gains in unique coverage finding ability over Syzkaller (4.28×), MoonShine (3.93×), ECG (2.22×), and KernelGPT (1.88×). It also surpasses these fuzzers in the number of valuable inputs ($N_{vi}$) by 36.27%, 36.26%, 14.24%, and 19.82%, respectively. In terms of syscalls metrics, SyzGPT covers 12.75%, 14.31%, 57.68%, 15.52%, 7.92%, 9.39%, and 24.24% more syscalls compared with all other fuzzers, with increases of 112.72%, 44.69%, 132.63%, 29.91%, 8.25%, 31.18%, and 46.60% in LFSC, respectively. The growths of $Cov$ and $N_s$ of SyzGPT and other fuzzers are shown in Figure 8.

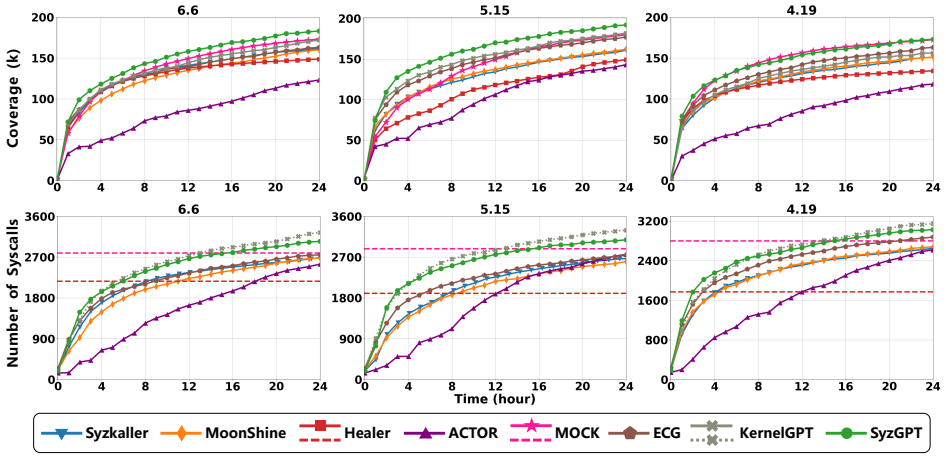


Fig. 8. Growth of the average coverage and syscalls of SyzGPT on three kernels over 24 hours compared to others. Since Healer and MOCK do not support recording syscalls over time, we plot their final numbers in dashed lines. We also plot the number of syscalls of KernelGPT in dashed line as it has higher $N_{enabled}$.

Averagely, MOCK ranks second in $Cov$, $R_s$, and $N_l$. This can be attributed to its context-aware mutation strategy, indicating that addressing dependency problem in kernel is effective for fuzzing. KernelGPT performs the best $N_s$ and the second best in $R_u$ and $Cov_u^l$, thanks to its new syscalls defined in Syzlang that enable it to cover unique code branches. However, its low $R_s$ indicates

the presence of the LFSC issue. ECG ranks second in $N_{vi}$, with good performance in *Cov* and $R_s$. This is likely due to its richer and LFSC-focused initial corpus, confirming that high-quality seed generation for LFSC enhances fuzzing outcomes. ACTOR ranks third in $N_l$, likely benefiting from its action-guided strategy, but this comes at the cost of lower *Cov*. These observations highlight the superiority of SyzGPT, which unlocks more syscalls (particularly LFSC) and achieves higher coverage by providing richer, high-quality seeds and more comprehensive dynamic syscall dependencies for kernel fuzzing. In addition, our experiments (as discussed in section 6) indicate that transferring our method to KernelGPT (MOCK as well) can further improve fuzzing performance.

**Vulnerability Detection.** We compare the vulnerability detection ability of SyzGPT to state-of-the-art fuzzers by analyzing the total crashes found during five repeats of 24-hour fuzzing. We ignore the errors introduced by fuzzer itself and de-duplicate the vulnerabilities according to the crash title. Figure 9 is an UpSet diagram [38] showing the intersection of the vulnerabilities found by eight fuzzers. On three LTS kernels, SyzGPT found 115 vulnerabilities, while Syzkaller, MoonShine, Healer, ACTOR, Mock, ECG, and KernelGPT found 78, 85, 12, 36, 11, 65, and 64, respectively. SyzGPT gains 47.44%, 35.29%, 858.33%, 219.44%, 945.45%, 76.92%, and 79.69% improvements in vulnerability detection compared to these fuzzers. And among all the vulnerabilities found on three kernels, SyzGPT found 27 unique ones, while Syzkaller, MoonShine, Healer, ACTOR, Mock, ECG, and KernelGPT only found 9, 14, 2, 10, 5, 9, and 13.

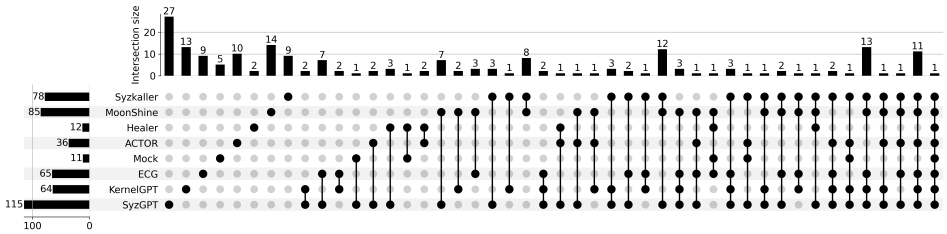


Fig. 9. Intersection of vulnerabilities discovered by SyzGPT and others of 5-repeat fuzzing on three kernels. The horizontal direction represents the total number and distribution of vulnerabilities discovered by each fuzzer, while the vertical direction shows which fuzzers identified the specific vulnerabilities.

We further analyze the relationship between LFSC and discovered vulnerabilities. We define vulnerabilities directly caused by LFSC as those including invocations of LFSC in crash call trace or Proof of Concept (PoC), while vulnerabilities indirectly caused by LFSC are difficult to evaluate because the seeds generated by SyzGPT for LFSC may also allow fuzzers to explore more code and discover more crashes in other branches at the same time. Therefore, we only count the vulnerabilities directly induced by LFSC (the actual amount would be higher). On three kernels, SyzGPT, Syzkaller, MoonShine, Healer, ACTOR, Mock, ECG, and KernelGPT discover 10, 2, 4, 6, 1, 4, 4, and 4 unique LFSC-introduced vulnerabilities, respectively. This indicates that SyzGPT is capable and effective in LFSC-related vulnerability detection as well.

### 5.3 RQ3: Ablation Study

**Ablation Study on Seed Generation.** We conduct an ablation study on seed generation for five relevant components: ❶ Syntax learning, ❷ Syscall dependency, ❸ DRAG context, ❹ Existing corpus, and ❺ Program repair. The baseline group is the original SyzGPT in default seed generation settings without program repair, as we need to compare the original outputs of different ablations on SyzGPT instead of the calibrated outputs. Notably, the re-generation strategy is also not included, as it is enabled during fuzzing.

Table 4. Contributions of each component of SyzGPT to the metrics of program generation with GPT-3.5, with baseline and best values highlighted in bold. Only Syzlang: dependencies extracted from Syzlang. Only Manpage: dependencies extracted from manpages. Fixed: fixed shots as context. Random: random shots as context. Regen: failure history as context.

(# / # exhibit the metrics without / with program repair.)

| Metrics | **SyzGPT** | System Prompt | Syscall Dependency | | Context | | | Existing Corpus | |
|---|---|---|---|---|---|---|---|---|---|
| LLM: GPT-3.5 | **Baseline** | Without Syntax | Only Syzlang | Only Manpage | Fixed | Random | Base + Re-gen | Syzbot | Local-120h |
| SVR (% / %) | **63.76** / 87.84 | 66.28 / 88.76 | 60.32 / 85.78 | 59.63 / 87.38 | 26.83 / 81.89 | 42.63 / 83.17 | **89.68** / 91.28 | 66.65 / 86.92 | 61.24 / **91.98** |
| CER (% / %) | **45.32** / 48.56 | 21.45 / 30.23 | 34.60 / 42.25 | 48.46 / 49.87 | 29.06 / 50.14 | 42.16 / 51.52 | 45.52 / 49.25 | **47.55** / **49.87** | 37.83 / 44.14 |
| OER (% / %) | **28.90** / 42.66 | 14.22 / 26.83 | 20.87 / 36.24 | 28.90 / 43.58 | 7.80 / 41.06 | 17.97 / 42.85 | **40.82** / **44.96** | 31.69 / 43.35 | 23.17 / 40.60 |
| $\overline{L}$ | **3.13** / 3.91 | 1.72 / 2.43 | 2.22 / 3.24 | 2.80 / 3.35 | 2.80 / 4.97 | 2.85 / 3.74 | **3.97** / **3.99** | 3.01 / 2.99 | 2.84 / 3.59 |
| $\overline{N_s}$ | **2.72** / 3.12 | 1.58 / 1.92 | 2.02 / 2.65 | 2.46 / 2.73 | 2.61 / 3.84 | 2.40 / 2.75 | **3.12** / **3.13** | 2.63 / 2.62 | 2.27 / 2.76 |
| Tokens | **1769.33** | 1121.65 | 1612.21 | 1955.37 | 1887.86 | 1602.25 | 1860.37 | 1752.00 | 1733.53 |
| Time (s) | **4.74** | 3.89 | 5.23 | 9.51 | 5.60 | 7.48 | 5.35 | 9.72 | 5.18 |

According to the System Prompt segment of Table 4, we find the absence of program syntax can keep the SVR but significantly reduce CER (from 45.32% to 21.45%), $\overline{L}$ (from 3.13 to 1.72) and $\overline{N_s}$ (from 2.72 to 1.58). We speculate that this is because, without syntax learning, LLM tends to mimic examples in the context rather than truly understanding the syntax of Syz-programs, resulting in programs that appear correct but lack genuine compliance. Compared to specialized call level dependencies from Syzlang, system call level dependencies from Manpage have close SVR and higher CER (from 34.60% to 48.46%). But they are all lower than the CER (45.32%) of baseline group with merged dependencies, proving the effectiveness of our syscall dependency augmentation. Compared to different ways of context construction, using fixed context results in obvious degradation in SVR (26.83%) and CER (29.06%) due to the limitation of sampled programs. Switching to random context (representing the absence of syscall dependency) can partially recover SVR (42.63%) and CER (42.16%), since there is a chance of randomly retrieving the appropriate R-programs. This demonstrates that our design of dependency-based R-programs retrieval is essential to effective seed generation. Additionally, re-generation based on baseline can improve all metrics. Regarding the existing corpus, utilizing Syzbot corpus enhances SVR, CER, and OER due to its richer set of programs and better contextual knowledge. The 120-hour local corpus yields a higher SVR than the baseline group, but slightly reduces other metrics. This may be because expanding the corpus scale will also increase the randomness of the R-programs retrieval, as their candidates are also growing. As for ablation of program repair, the results indicate that it significantly improves seed generation performance across all metrics. Finally, in terms of variation of OER compared to baseline, we can rank the contributions as: DRAG context (OER-21.10), syntax learning (OER-14.68), program repair (OER+13.76), syscall dependency (OER-10.93), existing corpus (OER+2.79).

**Ablation Study on Fuzzing.** Based on the previous ablation study, the context used in retrieval-augmented in-context learning and the program repair significantly contribute to seed generation performance. Therefore, we conduct the ablation study on fuzzing performance, focusing primarily on the following components: ❶ DRAG Context, ❷ Program repair, ❸ Base LLM, and ❹ Feedback-guided Seed Generation, which is only enabled in fuzzing.

According to Table 5, we can find that directly using LLM (Zero-shot) in kernel fuzzing results in the worst performance. This shows that without our method, the low-quality seeds generated by LLM itself waste fuzzing resources instead of improving performance. Compared with SyzGPT-Default, using the seeds generated for LFSC as initial corpus (SyzGPT-Init) increases LFSC-related syscall coverage ($R_s$ and $N_l$), but significantly reduces code coverage. While the initial corpus contains rich LFSC-related seeds that facilitate syscall coverage, it lacks runtime corpus knowledge, which limits the soundness for code coverage. This highlights the effectiveness of our feedback-guided seed generation strategy. Removing program repair (SyzGPT-NoRepair) and using random

Table 5. Ablation study of each SyzGPT component's contribution to the 24-hour fuzzing performance on 6.6, where DEC is the decrease compared to baseline SyzGPT-Default.

| Ablation Group | $Cov$ | ←DEC | $Cov_u$ / $R_u$ | $Cov_u^l$ | $N_{vi}$ | $N_s$ / $R_s$ | ←DEC | $N_l$ |
|---|---|---|---|---|---|---|---|---|
| Syzkaller | 163370 | -11.55% | *baseline* | 7110 | 29084 | 2678 / 68.70% | -11.99% | 278 |
| Zero-shot | 147941 | -19.90% | 14041 / 0.44 | 5826 | 20167 | 2385 / 61.19% | -21.62% | 274 |
| SyzGPT-Init | 172092 | -6.83% | 28610 / 1.98 | 7407 | 35246 | **3202 / 82.14%** | 5.23% | **971** |
| SyzGPT-Random | 173518 | -6.06% | 26197 / 1.48 | 7339 | 35977 | 2787 / 73.99% | -8.41% | 448 |
| SyzGPT-NoRepair | 175995 | -4.72% | 27139 / 1.70 | 7495 | 34163 | 2995 / 74.53% | -1.58% | 577 |
| SyzGPT-CodeLlama | 181778 | -1.58% | 30988 / 2.22 | 7548 | 34497 | 2911 / 72.81% | -4.34% | 521 |
| SyzGPT-CodeLlama-syz | **185809** | 0.60% | 32547 / 2.62 | **8329** | **37656** | 3110 / 79.78% | 2.20% | 593 |
| SyzGPT-Default | 184705 | - | **33960 / 3.23** | 8295 | 36871 | 3043 / 78.07% | - | 588 |

programs instead of R-programs retrieved by dependencies in context (SyzGPT-Random) also lead to significant reductions in all metrics, underscoring their indispensability for SyzGPT. Regarding using different base LLMs, CodeLlama achieves comparable code coverage to default GPT-3.5, indicating that our approach is scalable to LLMs with specialized programming knowledge. Furthermore, CodeLlama-syz shows even higher metrics except for $R_u$), implying that our fine-tuning based on LFSC-related program datasets can promote coverage, especially LFSC-related coverage, during fuzzing. The lower $Cov_u$ and $R_u$ compared to SyzGPT-Default likely stem from reduced output diversity due to instruction tuning. Finally, in terms of variation of $Cov$ compared to baseline, we can rank the contributions as: feedback-guided seed generation ($Cov$-6.83%), DRAG context ($Cov$-6.06%), and program repair ($Cov$-4.72%). The coverage and syscalls growth over time are plotted in Figure 10 for more intuitive understanding.

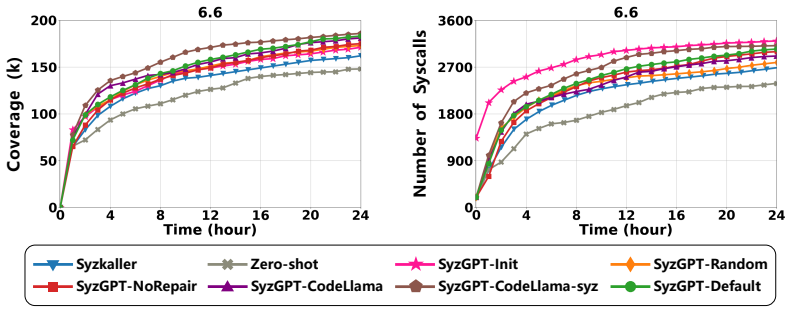


Fig. 10. Coverage and syscall growth of SyzGPT with different components disabled over 24 hours on average

## 5.4 RQ4: Real-World Bug Discovery

To evaluate the real-world bug discovery capabilities of SyzGPT, we conduct persistent fuzzing on the LTS kernel 6.6, 5.15, and 4.19, as well as several other versions during our development. As listed in Table 6, SyzGPT has independently found 26 unknown bugs during our experiments, 10 of which are directly related to LFSC and are hard to be discovered by other tools (6 more are indirectly related to LFSC through our crash log analysis). All bugs have been reported to upstream or downstream vendors, with 11 confirmed (including the fixed ones).

**Case Study.** We examine an LFSC-introduced 0-day bug found by SyzGPT. Specifically, the `iommufd_object` is allocated by one thread with `iommufd_ioas_alloc` and freed by another thread with `iommufd_destroy`. When another thread accesses the freed object by `iommufd_put_object`, it triggers use-after-free (UAF). By analyzing the bug's PoC (shown in the following listing), we find that the last three syscalls in the PoC are LFSC. Finally, we inspect the enriched corpus of the

fuzzer and find that each of the LFSC has several valid seeds generated by SyzGPT, demonstrating the effectiveness of SyzGPT in finding real-world bugs by addressing the problem of LFSC.

```
openat$iommufd(..BLINDED..)
ioctl$IOMMU_IOAS_ALLOC(..BLINDED..)
ioctl$IOMMU_TEST_OP_ADD_RESERVED(..BLINDED..) (async, rerun: BLINDED)
ioctl$IOMMU_DESTROY$ioas(..BLINDED..)
```

Table 6. New bugs independently discovered by SyzGPT during our experiments

| Kernel | Crash Type | Bug Function* | LFSC-related | PoC | Status |
|---|---|---|---|---|---|
| 6.6 | task hung | block_read_full_folio | ● | Syz,C | Reported |
| 6.6 | kernel bug | __gfs2_glock_put | ● | No | Reported |
| 6.6 | memory leak | wg_packet_send_keepalive | ◐ | No | Reported$^C$ |
| 6.6 | out-of-bounds | __lock_acquire | ○ | No | Reported |
| 6.6 | out-of-bounds | mceusb_set_tx_mask | ● | C | Confirmed$^C$ |
| 6.6 | use-after-free | iommufd_test | ● | Syz,C | Fixed$^C$ |
| 6.6 | out-of-bounds | udf_close_lvid | ● | No | Fixed |
| 6.6 | kernel bug | ntfs_collate_ntofs_ulong | ○ | Syz,C | Confirmed$^C$ |
| 6.6 | out-of-bounds | udf_finalize_lvid | ◐ | No | Reported |
| 6.6 | use-after-free | ntfs_attr_find | ○ | No | Reported |
| 6.6 | use-after-free | ccid2_hc_tx_packet_recv | ● | No | Reported |
| 6.6 | out-of-bounds | jfs_readdir | ○ | No | Fixed |
| 6.6 | general protection fault | tomoyo_profile | ● | Syz,C | Reported |
| 6.6 | general protection fault | bio_associate_blkg_from_css | ● | Syz,C | Fixed$^C$ |
| 6.5 | out-of-bounds | do_journal_end | ○ | Syz,C | Fixed |
| 6.5 | out-of-bounds | check_igot_inode | ◐ | No | Fixed |
| 6.5 | kernel bug | unix_bind | ○ | Syz,C | Reported |
| 6.5 | rcu stall | sys_getdents64 | ○ | Syz,C | Reported |
| 6.4 | use-after-free | radix_tree_descend | ● | No | Fixed |
| 6.2 | memory leak | p9_client_create | ◐ | Syz | Fixed$^C$ |
| 5.15 | general protection fault | __mod_lruvec_page_state | ○ | No | Reported |
| 5.15 | task hung | __rq_qos_throttle | ● | Syz,C | Reported |
| 5.15 | kernel bug | scsi_host_alloc | ◐ | Syz,C | Confirmed |
| 4.19 | use-after-free Read | nbd_put | ○ | No | Reported |
| 4.19 | use-after-free Read | hardware_disable | ○ | No | Reported |
| 4.19 | task hung | __jbd2_journal_force_commit | ◐ | Syz | Reported |

$^C$: Applying for CVE. ◐: Indirectly LFSC-related. * All bugs above were found by SyzGPT earlier than Syzbot.

## 6 Discussion

**Insights of LLMs for Kernel Fuzzing.** ❶ Defining program syntax is essential for niche programming languages. ❷ The context retrieved by RAG should adapt to the generated target, or it leads to superficial content imitation. ❸ The inconsistency in generation caused by LLM hallucinations is acceptable in kernel fuzzing, if we ensure validity. In addition to these insights, we summarize the optimizable subtasks of LLMs for kernel fuzzing in process order: specification generation, seed generation, seed mutation and scheduling, feedback mechanisms, vulnerability reproduction, and de-duplication. Currently, LLMs are only applied in the stages of specification generation (KernelGPT) and seed generation (ECG and SyzGPT). However, there is still valuable space that LLMs can utilize, such as execution traces for seed scheduling and call stacks for bug de-duplication.
**Transferability and Sustainability.** As a general framework for generating seeds for any syscalls in kernel fuzzing, SyzGPT can be seamlessly integrated with Syzkaller-like or syscall-based kernel fuzzers, enhancing their performance. For example, in syscall specification generation (e.g., SyzDescribe and KernelGPT), SyzGPT can effectively generate seeds for newly defined syscalls once they are included in the measurement scope. This is demonstrated by the improvements in SyzGPT-KernelGPT compared to KernelGPT (6.86% in *Cov* and 6.95% in $N_s$) as shown in Figure 11. Similarly, our approach can complement efforts like initial corpus optimization (e.g., MoonShine) and syscall dependency optimization (e.g., MOCK). SyzGPT can also assist directed kernel fuzzing once we obtain the mapping of target points to syscalls. The comparable results of SyzGPT-CodeLlama

indicates the adaptability of SyzGPT to different LLMs. Moreover, the superior results of SyzGPT-* over 72 hours in Figure 11 suggest that SyzGPT is not improving coverage by sacrificing the long-term potential. Instead, it enables more syscalls and enhances the coverage potential.

**Limitation and Future Work.** A small portion of programs generated by our method still have invalid syntax, due to complex syscall arguments that are difficult for LLMs to handle or incomplete repair operations. Such as memory offsets of variables like pointers or arguments with nested data structures. Therefore, we plan to design more comprehensive repair operations. As for the relatively low CER of our approach, one possible reason is that due to the retrieval randomness, the retrieved R-programs are not beneficial for the target syscall. We plan to propose a smarter R-programs retrieval algorithm by prioritizing the program candidates. Another possible reason is that some manpage documentation does not explain the dependencies between the syscalls and other related syscalls. We plan to pre-train LLMs based on more kernel documentation and effective Syz-programs in the future, which will fundamentally mitigate these limitations.

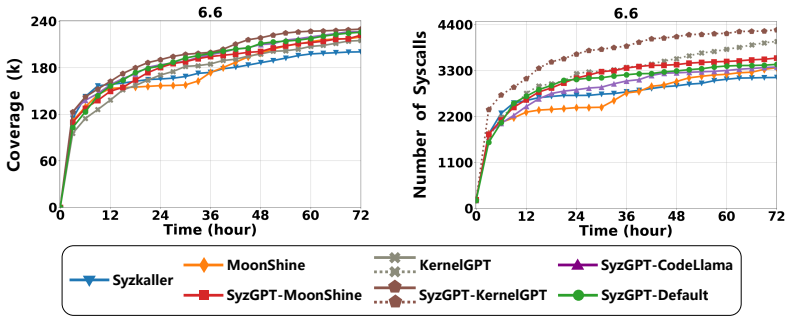


Fig. 11. The average 72-hour fuzzing results of the transferability of SyzGPT on different methods and LLM

## 7 Conclusion

In this paper, we design and implement SyzGPT, the first automated approach to generate programs via LLMs for low frequency syscalls (LFS) in kernel fuzzing. SyzGPT adopts a dependency-based retrieval-augmented generation (DRAG) method that leverages the historical corpus and syscall dependency knowledge to generate syntactically and contextually valid Syz-programs for LFS, which are hard to generate from fuzzer mutation or manual construction. With the periodically feedback-guided seed generation, SyzGPT improves code coverage by 31.85%, LFS syscall coverage by 17.73%, LFS coverage by 58.00%, and vulnerability detection by 323.22% over Syzkaller, MoonShine, Healer, ACTOR, MOCK, ECG, and KernelGPT on three LTS kernels, and independently finds 26 unknown bugs in Linux kernel (10 are LFS-related), with 11 confirmed.

## 8 Data Availability

Our artifact is maintained at https://github.com/QGrain/SyzGPT and archived in Zenodo [14].

## Acknowledgments

We thank the anonymous reviewers for their constructive feedback. This work is partially supported by NSFC (U24A20236, 62302497, 92270204), CAS Project for Young Scientists in Basic Research (Grant No. YSBR-118), and a research grant from Huawei.

## References


[1] 2023. Linux Kernel From First Principles. https://hackaday.com/2023/08/13/linux-kernel-from-first-principles/.
[2] 2024. CVE-2024-0565. https://nvd.nist.gov/vuln/detail/CVE-2024-0565.
[3] 2024. CVE-2024-0841. https://nvd.nist.gov/vuln/detail/CVE-2024-0841.
[4] 2024. CVE-2024-1086. https://nvd.nist.gov/vuln/detail/CVE-2024-1086.
[5] 2024. Linux Kernel Enriched Corpus. https://github.com/cmu-pasta/linux-kernel-enriched-corpus.
[6] 2024. Linux Kernel Syscall Table. https://syscalls.mebeim.net/?table=x86/64/x64/v6.6.
[7] 2024. Linux Manual Page, Section 2. https://man7.org/linux/man-pages/dir_section_2.html.
[8] 2024. Prog Deserialization. https://github.com/google/syzkaller/blob/master/prog/encoding.go#L255.
[9] 2024. Program Syntax. https://github.com/google/syzkaller/blob/master/docs/program_syntax.md.
[10] 2024. syz-execprog. https://github.com/google/syzkaller/blob/master/tools/syz-execprog/execprog.go.
[11] 2024. Syzbot. https://syzkaller.appspot.com/upstream.
[12] 2024. Syzkaller. https://github.com/google/syzkaller.
[13] 2024. Syzlang. https://github.com/google/syzkaller/blob/master/docs/syscall_descriptions_syntax.md.
[14] 2025. SyzGPT Artifact. https://zenodo.org/records/15537270. doi:10.5281/zenodo.15537270
[15] 2025. SyzGPT Repository. https://github.com/QGrain/SyzGPT.
[16] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023). doi:10.48550/ARXIV.2303.08774
[17] Owura Asare, Meiyappan Nagappan, and N Asokan. 2023. Is GitHub's Copilot as bad as humans at introducing vulnerabilities in code? *Empirical Software Engineering* 28, 6 (2023), 1–24. doi:10.1007/S10664-023-10380-1
[18] Erin Avllazagaj. 2023. SyzGPT: When the fuzzer meets the LLM. https://albocoder.github.io/fuzzing/exploitation/linux%20kernel/hacking/ai/gpt/llm/2023/11/27/GPT-syzkaller.html.
[19] Emily M Bender, Timnit Gebru, Angelina McMillan-Major, and Shmargaret Shmitchell. 2021. On the Dangers of Stochastic Parrots: Can Language Models Be Too Big?. In *Proceedings of the 2021 ACM conference on fairness, accountability, and transparency (FAccT '21)*. 610–623. doi:10.1145/3442188.3445922
[20] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language Models are Few-Shot Learners. *Advances in Neural Information Processing Systems* 33 (2020), 1877–1901. doi:10.48550/arXiv.2005.14165
[21] Alexander Bulekov, Bandan Das, Stefan Hajnoczi, and Manuel Egele. 2023. No Grammar, No Problem: Towards Fuzzing the Linux Kernel without System-Call Descriptions. In *Proceedings of the 30th Annual Network and Distributed System Security Symposium (NDSS)*. doi:10.14722/ndss.2023.24688
[22] Tianshi Cao, Marc Law, and Sanja Fidler. 2020. A Theoretical Analysis of the Number of Shots in Few-Shot Learning. In *8th International Conference on Learning Representations (ICLR)*. doi:10.48550/arXiv.1909.11722
[23] Weiteng Chen, Yu Hao, Zheng Zhang, Xiaochen Zou, Dhilung Kirat, Shachee Mishra, Douglas Schales, Jiyong Jang, and Zhiyun Qian. 2024. SyzGen++: Dependency Inference for Augmenting Kernel Driver Fuzzing. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 4661–4677. doi:10.1109/SP54263.2024.00269
[24] Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. 2023. Vicuna: An Open-Source Chatbot Impressing GPT-4 with 90%* ChatGPT Quality. https://lmsys.org/blog/2023-03-30-vicuna/.
[25] 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)*. 423–435. doi:10.1145/3597926.3598067
[26] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. 2024. Large Language Models are Edge-Case Generators: Crafting Unusual Programs for Fuzzing Deep Learning Libraries. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13. doi:10.1145/3597503.3623343
[27] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*. 4171–4186.
[28] Marius Fleischer, Dipanjan Das, Priyanka Bose, Weiheng Bai, Kangjie Lu, Mathias Payer, Christopher Kruegel, and Giovanni Vigna. 2023. ACTOR: Action-Guided Kernel Fuzzing. In *32nd USENIX Security Symposium (USENIX Security 23)*. 5003–5020. doi:10.5555/3620237.3620517
[29] Yu Hao, Guoren Li, Xiaochen Zou, Weiteng Chen, Shitong Zhu, Zhiyun Qian, and Ardalan Amiri Sani. 2023. SyzDescribe: Principled, Automated, Static Generation of Syscall Descriptions for Kernel Drivers. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 3262–3278. doi:10.1109/SP46215.2023.10179298
[30] Yu Hao, Hang Zhang, Guoren Li, Xingyun Du, Zhiyun Qian, and Ardalan Amiri Sani. 2022. Demystifying the Dependency Challenge in Kernel Fuzzing. In *Proceedings of the IEEE/ACM 44th International Conference on Software*

*Engineering*. 659–671. doi:10.1145/3510003.3510126

[31] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. 2022. LoRA: Low-Rank Adaptation of Large Language Models. In *10th International Conference on Learning Representations (ICLR)*. doi:10.48550/arXiv.2106.09685

[32] Dae R Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. 2019. Razzer: Finding Kernel Race Bugs through Fuzzing. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 754–768. doi:10.1109/SP.2019.00017

[33] Dae R Jeong, Byoungyoung Lee, Insik Shin, and Youngjin Kwon. 2023. SegFuzz: Segmentizing Thread Interleaving to Discover Kernel Concurrency Bugs through Fuzzing. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2104–2121. doi:10.1109/SP46215.2023.10179398

[34] Dave Jones. 2024. Trinity. https://github.com/kernelslacker/trinity. Accessed: 2024-01-06.

[35] Kyungtae Kim, Taegyu Kim, Ertza Warraich, Byoungyoung Lee, Kevin RB Butler, Antonio Bianchi, and Dave Jing Tian. 2022. FuzzUSB: Hybrid Stateful Fuzzing of USB Gadget Stacks. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2212–2229. doi:10.1109/SP46214.2022.9833593

[36] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. 7871. doi:10.18653/v1/2020.acl-main.703

[37] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems* 33 (2020), 9459–9474.

[38] Alexander Lex, Nils Gehlenborg, Hendrik Strobelt, Romain Vuillemot, and Hanspeter Pfister. 2014. UpSet: Visualization of Intersecting Sets. *IEEE Transactions on Visualization and Computer Graphics* 20, 12 (2014), 1983–1992.

[39] Raymond Li, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, LI Jia, Jenny Chim, Qian Liu, et al. 2023. StarCoder: may the source be with you! *Transactions on Machine Learning Research* (2023). doi:10.48550/arXiv.2305.06161

[40] Yuwei Li, Yuan Chen, Shouling Ji, Xuhong Zhang, Guanglu Yan, Alex X Liu, Chunming Wu, Zulie Pan, and Peng Lin. 2023. G-Fuzz: A Directed Fuzzing Framework for gVisor. *IEEE Transactions on Dependable and Secure Computing* 21, 1 (2023), 168–185. doi:10.1109/TDSC.2023.3244825

[41] Ruijie Meng, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury. 2024. Large Language Model guided Protocol Fuzzing. In *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS)*.

[42] Meta. 2024. Llama 3: The most capable openly available LLM to date. https://ai.meta.com/blog/meta-llama-3/.

[43] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. In *11th International Conference on Learning Representations (ICLR)*.

[44] OpenAI. 2022. Introducing ChatGPT. https://openai.com/blog/chatgpt.

[45] Shankara Pailoor, Andrew Aday, and Suman Jana. 2018. MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation. In *27th USENIX Security Symposium (USENIX Security 18)*. 729–743.

[46] Anthropic PBC. 2024. Claude 3.5 Sonnet. https://www.anthropic.com/news/claude-3-5-sonnet.

[47] Anthropic PBC. 2024. Introducing the next generation of Claude. https://www.anthropic.com/news/claude-3-family.

[48] Hui Peng and Mathias Payer. 2020. USBFuzz: A Framework for Fuzzing USB Drivers by Device Emulation. In *29th USENIX Security Symposium (USENIX Security 20)*. 2559–2575.

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

[50] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code Llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023). doi:10.48550/ARXIV.2308.12950

[51] Bonan Ruan, Jiahao Liu, Chuqi Zhang, and Zhenkai Liang. 2024. KernJC: Automated Vulnerable Environment Generation for Linux Kernel Vulnerabilities. In *Proceedings of the 27th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. 384–402. doi:10.1145/3678890.3678891

[52] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. 2017. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *26th USENIX security symposium (USENIX Security 17)*. 167–182.

[53] SecurityScorecard. 2024. CVE Details. https://www.cvedetails.com/vulnerability-list/vendor_id-33/Linux.html.

[54] Heyuan Shi, Shijun Chen, Runzhe Wang, Yuhan Chen, Weibo Zhang, Qiang Zhang, Yuheng Shen, Xiaohai Shi, Chao Hu, and Yu Jiang. 2024. Industry Practice of Directed Kernel Fuzzing for Open-source Linux Distribution. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 2159–2169.

[55] Hao Sun, Yuheng Shen, Jianzhong Liu, Yiru Xu, and Yu Jiang. 2022. KSG: Augmenting Kernel Fuzzing with System Call Specification Generation. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 351–366.

[56] Hao Sun, Yuheng Shen, Cong Wang, Jianzhong Liu, Yu Jiang, Ting Chen, and Aiguo Cui. 2021. Healer: Relation Learning Guided Kernel Fuzzing. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 344–358. doi:10.1145/3477132.3483547

[57] Xin Tan, Yuan Zhang, Jiadong Lu, Xin Xiong, Zhuang Liu, and Min Yang. 2023. SyzDirect: Directed Greybox Fuzzing for Linux Kernel. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 1630–1644. doi:10.1145/3576915.3623146

[58] Gemini Team, Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, et al. 2023. Gemini: A Family of Highly Capable Multimodal Models. *arXiv preprint arXiv:2312.11805* (2023). doi:10.48550/ARXIV.2312.11805

[59] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is All You Need. *Advances in Neural Information Processing Systems* 30 (2017).

[60] Dawei Wang, Geng Zhou, Li Chen, Dan Li, and Yukai Miao. 2024. ProphetFuzz: Fully Automated Prediction and Fuzzing of High-Risk Option Combinations with Only Documentation via Large Language Model. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. 735–749. doi:10.48550/arXiv.2409.00922

[61] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-Driven Seed Generation for Fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 579–594. doi:10.1109/SP.2017.23

[62] Yue Wang, Hung Le, Akhilesh Gotmare, Nghi Bui, Junnan Li, and Steven Hoi. 2023. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 1069–1088. doi:10.18653/V1/2023.EMNLP-MAIN.68

[63] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. *Advances in Neural Information Processing Systems* 35 (2022), 24824–24837. doi:10.48550/arXiv.2201.11903

[64] Wikipedia. 2024. Inverted index. https://en.wikipedia.org/wiki/Inverted_index.

[65] 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 IEEE/ACM 46th International Conference on Software Engineering*. 1–13. doi:10.1145/3597503.3639121

[66] Jiacheng Xu, Xuhong Zhang, Shouling Ji, Yuan Tian, Binbin Zhao, Qinying Wang, Peng Cheng, and Jiming Chen. 2024. MOCK: Optimizing Kernel Fuzzing Mutation with Context-aware Dependency. In *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS)*. doi:10.14722/ndss.2024.23131

[67] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. 2019. Fuzzing File Systems via Two-dimensional Input Space Exploration. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 818–834.

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

[69] Chenyuan Yang and Aleksandr Nogikh. 2024. sys/linux: add the descriptions for the CEC device . https://github.com/google/syzkaller/commit/d0304e9.

[70] Chenyuan Yang, Zijie Zhao, and Lingming Zhang. 2025. KernelGPT: Enhanced Kernel Fuzzing via Large Language Models. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '25)*. 560–573. doi:10.1145/3676641.3716022

[71] Ming Yuan, Bodong Zhao, Penghui Li, Jiashuo Liang, Xinhui Han, Xiapu Luo, and Chao Zhang. 2023. DDRace: Finding Concurrency UAF Vulnerabilities in Linux Drivers with Directed Fuzzing. In *32nd USENIX Security Symposium (USENIX Security 23)*. 2849–2866.

[72] Qiang Zhang, Yuheng Shen, Jianzhong Liu, Yiru Xu, Heyuan Shi, Yu Jiang, and Wanli Chang. 2024. ECG: Augmenting Embedded Operating System Fuzzing via LLM-Based Corpus Generation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 43, 11 (2024), 4238–4249. doi:10.1109/TCAD.2024.3447220

[73] Bodong Zhao, Zheming Li, Shisong Qin, Zheyu Ma, Ming Yuan, Wenyu Zhu, Zhihong Tian, and Chao Zhang. 2022. StateFuzz: System Call-Based State-Aware Linux Driver Fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*. 3273–3289.

[74] Penghao Zhao, Hailin Zhang, Qinhan Yu, Zhengren Wang, Yunteng Geng, Fangcheng Fu, Ling Yang, Wentao Zhang, and Bin Cui. 2024. Retrieval-augmented generation for ai-generated content: A survey. *arXiv preprint arXiv:2402.19473* (2024). doi:10.48550/ARXIV.2402.19473

[75] Daniel M Ziegler, Nisan Stiennon, Jeffrey Wu, Tom B Brown, Alec Radford, Dario Amodei, Paul Christiano, and Geoffrey Irving. 2019. Fine-tuning Language Models from Human Preferences. *arXiv preprint arXiv:1909.08593* (2019).

[76] Xiaochen Zou, Guoren Li, Weiteng Chen, Hang Zhang, and Zhiyun Qian. 2022. SyzScope: Revealing High-Risk Security Impacts of Fuzzer-Exposed Bugs in Linux kernel. In *31st USENIX Security Symposium (USENIX Security 22)*. 3201–3217.

Received 2024-10-31; accepted 2025-03-31