KernelGPT: Enhanced Kernel Fuzzing via Large Language Models

Chenyuan Yang
University of Illinois at Urbana-Champaign
Champaign, USA
cy54@illinois.edu

Zijie Zhao
University of Illinois at Urbana-Champaign
Champaign, USA
zijie4@illinois.edu

Lingming Zhang
University of Illinois at Urbana-Champaign
Champaign, USA
lingming@illinois.edu

Abstract

Bugs in operating system kernels can affect billions of devices and users all over the world. As a result, a large body of research has been focused on kernel fuzzing, i.e., automatically generating syscall (system call) sequences to detect potential kernel bugs or vulnerabilities. Kernel fuzzing aims to generate valid syscall sequences guided by syscall specifications that define both the syntax and semantics of syscalls. While there has been existing work trying to automate syscall specification generation, this remains largely manual work, and a large number of important syscalls are still uncovered.

In this paper, we propose KernelGPT, the first approach to automatically synthesizing syscall specifications via Large Language Models (LLMs) for enhanced kernel fuzzing. Our key insight is that LLMs have seen massive kernel code, documentation, and use cases during pre-training, and thus can automatically distill the necessary information for making valid syscalls. More specifically, KernelGPT leverages an iterative approach to automatically infer the specifications, and further debug and repair them based on the validation feedback. Our results demonstrate that KernelGPT can generate more new and valid specifications and achieve higher coverage than state-of-the-art techniques. So far, by using newly generated specifications, KernelGPT has already detected 24 new unique bugs in Linux kernel, with 12 fixed and 11 assigned with CVE numbers. Moreover, a number of specifications generated by KernelGPT have already been merged into the kernel fuzzer Syzkaller, following the request from its development team.

CCS Concepts: • **Security and privacy → Operating systems security**; • **Software and its engineering → Software testing and debugging**.

Keywords: Linux Kernel, Fuzzing, Large Language Models, Code Analysis

ASPLOS '25, March 30-April 3, 2025, Rotterdam, Netherlands

ACM ISBN 979-8-4007-1079-7/25/03
https://doi.org/10.1145/3676641.3716022

ACM Reference Format:
Chenyuan Yang, Zijie Zhao, and Lingming Zhang. 2025. KernelGPT: Enhanced Kernel Fuzzing via Large Language Models. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3676641.3716022

1 Introduction

Operating system kernels are among the most critical systems, as all other types of systems rely on and operate on them. Kernel vulnerabilities, such as crashes or out-of-bounds writes, can be maliciously exploited, potentially causing substantial harm to all users. To ensure the correctness and security of these fundamental systems, fuzzing (or fuzz testing) [49, 64, 67] has been employed for decades. Such techniques automatically generate a vast number of system calls as test inputs, intending to detect potential kernel bugs.

Among various kernel fuzzing techniques [23, 28, 39], Syzkaller [5] is one of the most popular tools. Syzkaller has identified over 5K bugs that are recognized and fixed by kernel developers [4]. Thus, numerous research efforts have focused on enhancing Syzkaller, targeting areas such as seed generation [40, 42], seed selection [55], guided mutation [21, 48], and syscall specification generation [14, 15, 25, 47]. Among these, the syscall specifications written in syzlang [6] are particularly crucial, significantly contributing to the effectiveness of Syzkaller and allowing it to cover more kernel modules. They specify the syntax of syscalls, and their intra- and inter-dependencies, enabling the generation of more valid syscall sequences that probe deeper into the kernel code logic. However, crafting syscall specifications is difficult because it is predominantly a manual process and require much in-depth kernel knowledge.

To address this issue, recent research has focused on automating the generation of syscall specifications, particularly for device drivers. For instance, DIFUSE [15] and SyzDescribe [25] employ static code analysis to identify device driver syscall handlers and infer their corresponding descriptions. The top half of Figure 1 illustrates the workflow of static analysis-based techniques. Initially, experts manually define rules to infer descriptions from the source code, drawing upon their own understanding of the kernel codebase

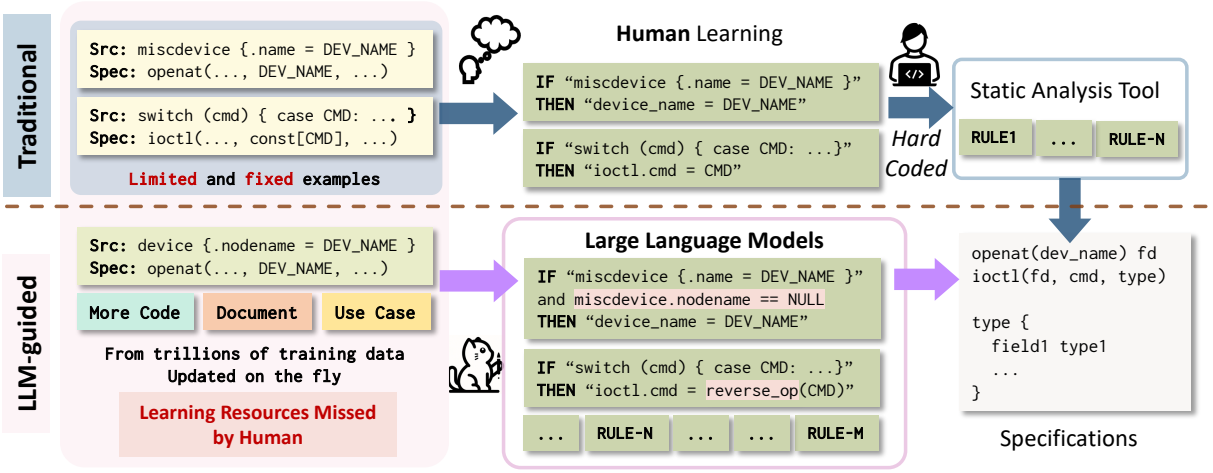

Figure 1. Workflows of syscall specification inference based on static analysis and LLMs

and existing Syzkaller examples. The accuracy and effectiveness of the generated syscall descriptions depend heavily on the comprehensiveness of these mapping rules, which is often challenging, costly, and tedious. Moreover, as the kernel codebase evolves, these mapping rules are subject to frequent changes. Keeping up with these evolving scenarios is a significant challenge for static analysis methods, particularly given the extensive implementation efforts involved. Plus, existing approaches struggle to generate human-readable specifications, yet readability is essential for validation and maintenance, according to Syzkaller developers [3].

Take, for instance, Figure 2a and Figure 2b, which illustrates the source code of two `struct` variables, associated with the device mapper driver [57], responsible for mapping physical block devices to higher-level virtual block devices. Specifically, these two variables are the device operation handler and its reference usage, crucial for inferring the device name. Current advanced syscall description generators, like SyzDescribe [25], typically rely on the field `name` in `struct miscdevice` to determine the device name for driver interaction, which is a conventional use case. However, in this example, the correct device name is actually specified in the field `nodename`, a legitimate but rare use case, leading to an incorrect inference by SyzDescribe. Moreover, it fails to analyze the command value for `ioctl`, the interface to interact with the device. This is because the command value undergoes a modification in the code, `cmd = _IOC_NR(command)`, where `command` is from users. Such scenarios are not accounted for by SyzDescribe, which erroneously uses `cmd` as the command value in its generated descriptions, as shown in Figure 2c.

Key insight. Can we automate and improve the learning of various rules for generating high-quality specifications from the codebase with minimal effort? We address this question based on the insight that modern Large Language Models (LLMs) [13, 20, 31, 37, 38, 44, 56] are pre-trained on vast

```
static const struct file_operations _ctl_fops = {
  .open = dm_open,
  .release = dm_release,
  .poll = dm_poll,
  .unlocked_ioctl = dm_ctl_ioctl,  IOCTL handler
  .compat_ioctl = dm_compat_ctl_ioctl,
  .owner = THIS_MODULE,
  .llseek = noop_llseek,
};
```

(a) Device operation handler

```
static struct miscdevice _dm_misc = {
  .minor = MAPPER_CTRL_MINOR,
  .name = DM_NAME,
  .nodename = DM_DIR "/" DM_CONTROL_NODE,
  .fops = &_ctl_fops
};
```

(b) Device operation handler reference

```
resource fd_34545[fd]                    Wrong device name
openat$34545(..., "/dev/device-mapper"]], ...) fd_34545
...                                      Wrong CMD value
ioctl$34545_2(fd fd_34545, cmd const[2], arg ptr[in, array[int8]])
# 5 descriptions are omitted             Inaccurate arg type
# No definition for dm_ioctl
```

(c) Specification generated by SyzDescribe

```
resource fd_dm[fd]
openat$dm(..., "/dev/mapper/control", ...) fd_dm
...
ioctl$DM_LIST_DEVICES(fd fd_dm, cmd const[DM_LIST_DEVICES],
arg ptr[inout, dm_ioctl])                CMD value: 3241737474
# 17 descriptions are omitted
dm_ioctl {
  version array[int32, 3]
  data_size int32 ⟶  "WARNING: kmalloc bug in ctl_ioctl"
  ...
}
```

(d) Specification generated by KernelGPT

Figure 2. Device mapper driver in `drivers/md/dm-ioctl.c`

datasets, including kernel codebases, documentation, and real-world syscall use cases. The Linux kernel's extensive history and associated wealth of discussions, tutorials, and documentation further enrich this training data. Consequently,

LLMs have likely been exposed to and potentially learned a wide range of information about syscall specifications. This superior knowledge base makes them adept at analyzing kernel source code, even for atypically implemented syscalls, and generating *high-quality, readable* specifications.

Utilizing LLMs as shown in Figure 1, we can automate the process of inferring rules for mapping codebase content to syscall specifications and tailor these rules to be more general and adaptable to diverse cases. Additionally, this approach eliminates the need for hard-coding rules within complex static analysis tools since LLMs inherently can analyze code, which significantly eases the process of adapting to evolving changes within the kernel codebase. Returning to the case of the device mapper driver, LLMs demonstrate their capability to accurately infer broader rules. They recognize that `.nodename` should be used as the device name when it is set and can identify modifications made to the command value. Consequently, in our experiments, the specification generated by LLMs for the device mapper (Figure 2d) is not only correct but also more complete compared to those produced by SyzDescribe. Impressively, this specification inferred by LLMs contributes to the discovery of 3 new bugs for this driver, 2 assigned with CVEs.

Building on the insight discussed above, we introduce KernelGPT, the first approach to fully automate syscall specification generation by using Large Language Models (LLMs), focusing on kernel drivers and sockets. The key idea of KernelGPT is to employ LLMs for automating and enhancing the rule inference process, aimed at synthesizing high-quality syscall descriptions from their source code. Taking the located operation handler as input, KernelGPT recovers the identifier value (*e.g.,* device name or command value), type structure, and dependency for the syscalls related to the handler. To this end, KernelGPT iteratively applies LLMs to analyze the relevant source code and indicate the missing but essential information for inference, which will be analyzed in the next iteration. Afterward, KernelGPT validates and repairs the generated specifications by consulting LLMs with the error messages encountered.

Our contributions are summarized below:

- We propose the first *automated* approach to leveraging the potential of LLMs for kernel fuzzing. Moreover, different from existing LLM-based fuzzing work [17, 59, 63], our approach goes beyond merely generating test inputs; we synthesize components of the fuzzing framework to integrate LLMs with matured frameworks developed for years, opening a new dimension for LLM-based fuzzing.
- We implement KernelGPT to infer syscall specifications with a novel iterative strategy and further repair the descriptions with the validation feedback. Our artifact is available at https://github.com/ise-uiuc/KernelGPT.
- We evaluate KernelGPT in generating new specifications to detect bugs and for the existing drivers and sockets to compare against state-of-the-art baselines, SyzDescribe and Syzkaller. Our experimental results show that KernelGPT can generate more new and valid syscall descriptions and achieve higher coverage than baselines.
- KernelGPT has already detected 24 previously unknown bugs, with 12 fixed and 11 CVE assignments, in the upstream Linux kernel. Notably, a number of specifications generated by KernelGPT are merged into Syzkaller, following a request from its development team.

2 Background

2.1 Kernel Fuzzing

OS Kernel Bugs. An OS kernel provides userspace applications with key functionalities, such as virtual memory, file system, networking, and access to devices. To protect the safety of all applications and users, interactions between userspace and kernel are confined to a well-defined system call interfaces (**syscall**), *e.g.,* the POSIX standard. Kernel bugs that can be triggered through the syscall interface pose a significant risk since the interface is easily accessible to attackers. Therefore, detecting bugs through the syscall interface has been an important direction of kernel security. In this work, we focus on detecting kernel bugs through the Linux kernel system call interface.

Device driver and socket. Device drivers and sockets are the most complex and important components in the Linux kernel, comprising 41.6% and 27% of the LoC, respectively [12]. Due to the diversity of devices and network protocols, the syscall for interacting with drivers and sockets is complex.

Drivers and sockets register syscall handlers that are invoked when corresponding syscalls are used. Device drivers communicate with hardware upon receiving syscalls. Figure 2a and Figure 2b show data structures used for registering drivers. The `nodename` field represents the device file name, and the `fops` field stores function pointers for custom handler functions. When a user calls `open` with the `nodename`, kernel invokes the `dm_open` handler and associates the file descriptor with the driver. Subsequent syscalls with this file descriptor invoke corresponding handlers. Sockets register syscalls like `socket`, `recvfrom`, and `setsockopt`. Each driver and socket registers different handlers, requiring unique specifications for effective fuzzing.

Generic syscalls. While drivers and sockets can register syscall handlers, only a limited number of syscalls can be registered, and they may not cover all necessary operations. Generic syscalls like **`ioctl`** and **`setsockopt`** are heavily used. They have numeric parameters (identifier values) and an untyped pointer parameter. The numeric parameter identifies the operation, and the untyped pointer is cast to the required data structure. For example, to get the list of `dm` device names, an application calls `ioctl` with the `DM_LIST_DEVICES` macro and a pointer to `struct dm_ioctl`. Sockets follow a similar pattern for programming `setsockopt` handlers.

The extensive usage of *syscall handlers* and *generic syscalls* turns a handful of syscalls into thousands of different syscalls, exposing a large attack surface. Even worse, the implementations are scattered across a wide array of drivers and sockets. This makes reasoning, analyzing, and testing the device drivers and sockets particularly challenging.

Syscall Fuzzer. Among various methods for kernel bug detection [14, 15, 19, 23–25, 28, 42, 48, 52, 55], Syzkaller [5], the state-of-the-art kernel fuzzer, has identified thousands of kernel bugs. Syzkaller uses the syntax and semantics of syscalls to generate diverse syscall sequences that can cover deep and diverse code paths. To define the syntax and semantics of syscalls, Syzkaller provides a domain-specific language, syzlang [6], to define **syscall specifications** (or descriptions). Figure 3 are example specifications of three syscalls for the MSM driver, showcasing syzlang's expressive power:

- *Syntax*: The syntax of a system call is expressed by the definition of parameter types. The types `int32`, `string`, `ptr`, and the struct `rm_msm_submitqueue` allow Syzkaller to know how to structure the bytes for all the parameters.
- *Semantically Valid Values*: Some parameters have specific value requirements. For example, the filename `"/dev/msm"` is the only valid file for the MSM driver.
- *Inter-Syscall Dependency*: One syscall may depend on the output of another. The return value of `openat$msm`, `fd_msm`, is the same variable as the first inputs of the two `ioctl` syscalls, indicating their sequential execution.
- *Intra-Syscall Dependency*: For generic syscalls like `ioctl`, the semantics of one parameter may depend on the value of another parameter. Syzkaller allows defining multiple specifications for the same syscall to express fine-grained semantics. For example, for `ioctl$NEW` and `ioctl$CLOSE`, the types of the third parameter depend on the macro value of the second.
- *Type Constraints*: Type definitions can incorporate semantic constraints. For example, in `rm_msm_submitqueue`, `[0:3]` represents the valid range of `prio`, and `(out)` denotes that `msm_submitqueue_id` is used as output.

While syzlang supports more advanced features, their core functionalities are similar to those discussed. Effective specifications enable Syzkaller to reduce the search space by filtering out invalid syscall sequences. However, creating these specifications requires a deep understanding of syscall semantics, challenging both humans and automated tools.

```
resource fd_msm[fd]
resource msm_submitqueue_id[int32]

openat$msm(..., file ptr[in, string["/dev/msm"]], ...) fd_msm
ioctl$NEW(fd fd_msm, cmd const[DRM_IOCTL_MSM_SUBMITQUEUE_NEW],
          arg ptr[inout, drm_msm_submitqueue])
ioctl$CLOSE(fd fd_msm, cmd const[DRM_IOCTL_MSM_SUBMITQUEUE_CLOSE],
            arg ptr[in, msm_submitqueue_id])
drm_msm_submitqueue {
  flags flags[msm_submitqueue_flags, int32]
  prio int32[0:3]
  id msm_submitqueue_id (out)
}
```

Figure 3. Specification for the MSM driver in syzlang

2.2 Specification Generation

Specifications are typically manually written by Syzkaller and kernel developers, requiring deep expertise in kernel and the specific kernel module. Thus, existing Syzkaller specifications only cover a subset of syscalls, especially for device drivers [12]. As the kernel evolves, specifications can become out-of-date [12, 25]. Automated specification generation has been desired for years [3], but faces several challenges. First, discovering the interface of the vast number of operations implemented behind generic syscalls is challenging. This involves inferring the correct operation identifier value (*e.g.,* device file name, socket domain, or `ioctl` command value), and then the corresponding data type for each unique operation. Moreover, finding the dependencies among syscalls and data types is also key to reaching deep paths. Failing to address these challenges would lead to inaccurate specifications, diminishing the effectiveness of a fuzzing campaign. Aside from fuzzing performance, readability of the machine-generated specification is also crucial for human experts to validate and maintain [3]. Unreadable specifications could hide flaws that in turn hurt the effectiveness of fuzzing.

Several techniques for specification generation have been proposed, attempting to address some of the above challenges. KSG [47] generates specifications by dynamically probing the kernel. It first opens the devices (or sockets) existing in a booted environment and probes the kernel to detect their syscall handlers. Then it infers the handler's parameter type through symbolic execution. Relying on existing device files, KSG is unable to generate specifications for drivers that are not loaded in the kernel or require more setup steps. In contrast, DIFUZE [15] and the state-of-the-art specification generation approach, SyzDescribe [25], both employ static analysis. DIFUZE finds syscall handlers from a list of data structures used by common device registration functions. SyzDescribe discovers syscall handlers by finding the kernel module initialization functions and tracing down to find the handler function pointers. Both DIFUZE and SyzDescribe then conduct static analysis to identify the device file name, command value, and required parameter type. Their static analysis models common implementation patterns *e.g.,* a switch case in a handler is likely invoking the corresponding sub-handlers based on the command value.

2.3 Challenges and Opportunities

Existing static analysis approaches to specification generation face several limitations.

L-1: Incomplete modeling. Rule-based approaches struggle to capture the diversity of kernel code patterns, leading to limited coverage. Maintaining these rules is challenging and impractical.

L-2: Readability. Static analysis often generates specifications that are difficult for humans to understand, hindering validation and maintenance [3].
L-3: Textual comprehension. These tools struggle to infer specifications from textual information, such as comments, limiting their ability to capture the underlying meaning and intent of syscall behavior.
Solution: Leveraging LLMs. To address the limitations, we propose a novel approach leveraging the strengths of LLMs:

- *Mitigating L-1*: LLMs are pre-trained on extensive codebases, enabling them to handle a broader range of cases more effectively than static analysis rules.
- *Mitigating L-2*: LLMs can generate descriptive and human-readable names within specifications based on code, enhancing readability and maintenance.
- *Mitigating L-3*: LLMs excel in interpreting textual information, producing specifications that capture the underlying meaning and intent of syscall behaviors.

While harnessing the potential of LLMs, we must design strategies to mitigate their inherent limitations, such as context size restrictions and hallucinations [27]. To achieve this, we 1) incorporate syz-lang knowledge through few-shot prompting, 2) develop a novel iterative multi-stage prompting approach, and 3) leverage off-the-shelf validation tools for debugging. More details will be discussed in § 3.

3 Design

Figure 4 presents the overview workflow of KernelGPT, which utilizes a code extractor and an analysis LLM (§ 4) to fully automatically generate specifications for kernel fuzzing.

KernelGPT takes the kernel codebase and located operation handlers as input and operates through two automated phases: Specification Generation ❶, and Specification Validation and Repair ❷. Initially, KernelGPT determines the identifier values (§ 3.1.1), argument types (§ 3.1.2), and dependencies (§ 3.1.3) for describing the syscalls associated with the given operation handler. In doing so, KernelGPT utilizes the relevant source code from the kernel codebase to guide the LLMs in their analysis in a novel iterative way. If essential information for inference is missing, the analysis LLM is instructed to indicate what additional information is required, which is then gathered and presented for analysis in the following step (§ 3.1 ❶). Subsequently, KernelGPT validates the generated specifications. If errors are found, it attempts to repair the descriptions by consulting the LLMs with the error messages (§ 3.2 ❷).

3.1 Specification Generation

KernelGPT generates the specifications for syscalls by leveraging LLMs to analyze the implementation source code of the syscalls. Initially, we identify the syscall handler functions (e.g. `ioctl` and `setsockopt`) from their respective operation handlers. We segment the specification generation process into three stages: identifier deduction, type recovery, and dependency analysis. This pipeline enables LLMs to focus on one specific aspect at each stage and avoid misleading information from irrelevant code snippets. In each stage, we utilize in-context few-shot prompting [9] to enhance LLMs' comprehension of the task and formalize output.

Algorithm 1: LLM-Guided Iterative Analysis

```
1  Function Analyze(relatedCode, usageInfo, step):
2      if step > MAX_ITER then
3          return ∅
4      # Prepare the prompt with few-shot examples
5      prompt ← GenPrompt (relatedCode, usageInfo)
6      # Query LLM to analyze the source code and identify the unknown
7      result, unknown ← QueryLLM (prompt)
8      for (id, usageInfo) ∈ unknown do
9          # Extract the code based on the unknown identifier
10         relatedCode ← ExtractCode (id)
11         # Recursively analyze the missing source code
12         res ← Analyze (relatedCode, usageInfo, step + 1)
13         # Update with the result analyzed for the unknown
14         Update (result, res)
15     return result
```

Iterative analysis. All three stages follow an iterative analysis paradigm. The motivation for this iterative design is two-fold. First, even though state-of-the-art LLMs like GPT4 can support long context size up to 128K [2], this size is still not enough to provide the entire source code related to a syscall. Second, the goal of specification generation is to deduce the identifier value, type structure, and dependency for the syscall. However, not all code or helper functions within the syscall handler are directly relevant to this goal. Consequently, we allow LLMs to identify the pertinent source code for the current goal.

The pipeline of the iterative analysis is shown in Algorithm 1. First, we generate a few-shot example prompt with the source code related to the target syscall and its usage information (Line 5). Then, we query LLMs to infer the descriptions and pinpoint the unknown targets (functions or types; Line 7). Such unknown functions/types are the ones that are missing in the provided prompt yet are essential for the inference. `unknown` set. Then, for each unknown target, we extract its source code by using its identifier. This code is then fed to LLMs along with its usage information for further analysis (Line 12). This iterative algorithm continues until there is no unknown target or the iteration reaches a predefined threshold `MAX_ITER` (Line 3). Note that the entire analysis process is *fully automated* and requires no human intervention. The unknown information provided by LLMs is used to guide the analysis process. Next, we demonstrate in detail how each stage employs this iterative algorithm.

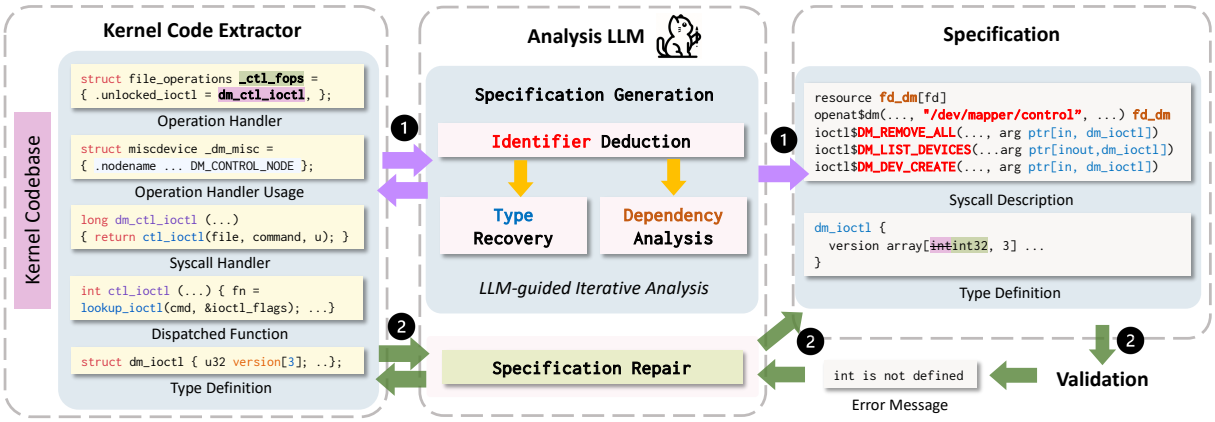

Figure 4. Overview of KernelGPT

3.1.1 Identifier Deduction.

The first step of specification generation is to deduce the identifier value of the syscall. To achieve this goal, we utilize the iterative strategy described in Algorithm 1 to analyze the syscall-related source code. The expected output from LLMs (the output of `QueryLLM`) is the set of successfully inferred identifier values (`result`). If the logic for checking identifier values is delegated to another function not presented to LLMs, we instruct LLMs to list the name and invocation details of this "missing" dispatched function (the variable `unknown`). Besides, we also include code snippets that reference the command variables. If LLMs identifies any unknown identifier values, KernelGPT proceeds to analyze the newly identified dispatched function, incorporating their usage information from the previous step. In essence, the output from the `unknown` of the previous step serves as a reference for guiding subsequent steps.

Compared to traditional static analysis, LLMs displays great potential to handle a wider range of scenarios in identifier value inference. To help with the LLMs understanding, we provide a few-shot example within the prompt (`GenPrompt`). These examples serve as guides for LLMs to improve their reasoning and deduce the identifier values more effectively.

3.1.2 Type Recovery.

Following identifier value inference, the next stage is to analyze the argument type structure for each identifier value. Leveraging information from the identifier deduction stage, we extract related functions and present them to the LLM to identify argument types. If type determination logic is delegated to other functions, we continue the analysis in subsequent steps, using the new information to guide the process. After determining the argument types, KernelGPT generates descriptions for these types. By retrieving type definition source code from the Linux kernel codebase and feeding it to the LLM, we obtain Syzkaller descriptions. If nested types are encountered, they are marked as `unknown` for further analysis in following steps.

While static analysis can recover type definitions from source code, it is difficult to infer the semantic relationships between nested types, particularly within `struct` and `union` definitions. For instance, consider the field `count` within the structure in Figure 5. `count` represents the number of elements within another field, `devices`. Traditional static analysis tools struggle to capture this inherent relationship and treat these fields independently. In contrast, LLMs can understand the semantic connections between different fields or types within nested structures. KernelGPT leverages this capability to generate descriptions that capture these relationships. As depicted in Figure 5, the description produced by KernelGPT is `count len[devices]`. The generated description can effectively capture the semantic connection between `count` and the number of elements in `devices`.

```
## Source Code
struct vfio_pci_hot_reset_info {
    int32   count;
    struct vfio_pci_dependent_device    devices[];
};

## Specification Generated by Static Analysis
vfio_pci_hot_reset_info {
  field_0 int32
  field_1 array[vfio_pci_dependent_device]
}

## Specification Generated by LLM
vfio_pci_hot_reset_info {
  count len[devices, int32]
  devices ptr[inout, array[vfio_pci_dependent_device]]
}
```

Figure 5. Type definitions from static analysis and LLM

3.1.3 Dependency Analysis.

Finally, we need to analyze the dependencies between syscalls. Specifically, the dependency means whether another syscall (or operation handler) relies on the return value of the current syscall. To achieve this goal, we leverage LLMs to identify if the return value could be a resource (*e.g.*, file descriptor) of another operation handler. KernelGPT extracts the source code of the relevant functions and presents the code to LLMs. Notably, the return value relevant functions are marked by LLMs themselves in

the first stage (§ 3.1.1). Suppose the return value can be used by other syscalls, LLMs identifies them. If the logic for dependency analysis resides in other functions, LLMs marks them as unknown, and KernelGPT utilizes the new information to continue the analysis in subsequent steps.

3.2 Specification Validation and Repair

In this phase, inspired by recent work on LLM-based program repair [61], our goal is to validate the specifications generated by KernelGPT and automatically repair the invalid ones. This is because LLMs may occasionally make mistakes during the description generation process. To address potential inaccuracies, we employ off-the-shelf validation tools. These tools analyze the specifications and provide error messages if discrepancies are found. Initially, KernelGPT uses the error messages from them to pinpoint inaccuracies in specific descriptions, effectively matching each error message to its corresponding description. Then, for those descriptions identified with errors, KernelGPT queries LLMs for correction, guided by few-shot examples. This process involves supplying LLMs with the incorrect description, the associated error messages, and relevant source code from the kernel codebase to repair. LLMs are then expected to output the correct descriptions.

Existing validation tools are limited in their ability to detect semantic errors [6], primarily focusing on syntax validation and simple semantic checks. For example, runtime validation of semantic correctness remains a significant challenge, which is why current syscall specification generation approaches do not incorporate it [15, 25]. To ensure a more thorough evaluation, we manually examined the generated specifications (§ 5.1.3), demonstrating that KernelGPT successfully synthesizes semantically correct specifications.

4 Implementation

Target. While our approach is general to various syscalls, KernelGPT targets those for kernel drivers and sockets, given the fact that they constitute about 70% LoC in the kernel [12]. For drivers, we focus on the critical `ioctl` syscall, in addition to initialization syscalls such as `openat` and `sys_open_dev`. For sockets, we extend our support beyond `socket` and `ioctl` to include syscalls like `bind`, `connect`, `accept`, `poll`, `sendto`, `recvfrom`, `setsockopt`, and `getsockopt`.

Source code extractor. It is implemented using the LLVM toolchain [7] and parses the kernel codebase to:

- *Driver and Socket Operation Handler Extraction.* The extractor employs *simple yet general* pattern matching to pinpoint driver and socket operation handlers. These are then prepared as inputs for KernelGPT, extracted with their corresponding usage locations. More specifically, we search for initialization instances of the `ioctl` or `unlocked_ioctl` fields within the operation handlers. For instance, the device mapper driver shown in Figure 2a initiates the `unlocked_ioctl` field in the structure `_ctl_fops` by using `dm_ctl_ioctl` function. We label `_ctl_fops` as the device operation handler and extract `dm_ctl_ioctl` to generate specifications for `ioctl` syscalls. KernelGPT focuses on inference from source code to descriptions, so we use a straightforward pattern-searching method to find device and socket operations.
- *Kernel Definition Extraction.* The extractor compiles all definitions of `function`, `struct`, `union`, and `enum` found within the kernel. These definitions are used as guidance for LLMs in the specification generation and repair processes, provided when LLMs indicate their necessity (**`ExtractCode`** function in Algorithm 1).

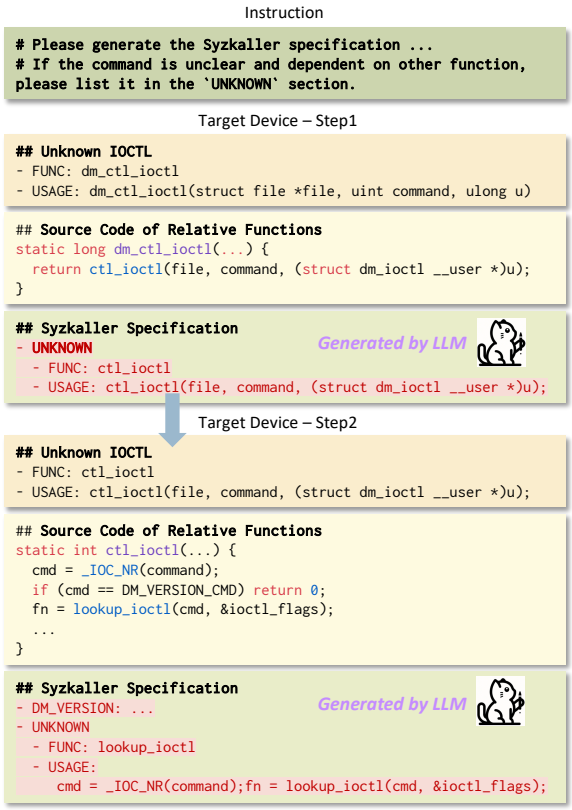

Figure 6. Iterative prompt for identifier deduction

Analysis LLM. While our approach is general and independent of the specific LLMs used, our tool, KernelGPT, is constructed atop GPT-4 [38]. At each step, we utilize the OpenAI APIs to query GPT-4, with a low-temperature of `0.1`. We set the stopping criteria for analysis as 5 by default.

Iterative analysis. We design a structured prompt template to facilitate interaction with LLMs for the kernel code analysis. For instance, Figure 6 presents the first two steps of identifier deduction (§ 3.1.1) for the device mapper driver.

The `dm_ctl_ioctl` handler, registered as the `ioctl` handler, offloads its entire functionality to another function, `ctl_ioctl`. As a result, after examining `dm_ctl_ioctl`, LLMs are unable to deduce any identifier values and designate `ctl_ioctl` as the absent function. Then KernelGPT extracts the source code for `ctl_ioctl` and, together with the unknown information returned by the first step, re-queries LLMs. In the second round, LLMs successfully identifies one identifier value, `DM_VERSION`, while other values related to the function `lookup_ioctl` remain undetermined. Thus, LLMs report `DM_VERSION` as one identifier value and `lookup_ioctl` as the missing function, which will be analyzed in the next step to infer more identifier values.

We set the default value of `MAX_ITER` to 5 (Line 3, Algorithm 1). For efficiency, our implementation caches and reuses results from previously explored paths. With these configurations, we observed no termination issues throughout our experiments.

Specification generation. Rather than generating descriptions for all possible drivers and sockets, we focus on those not covered by existing specifications, which are often less thoroughly tested. We select drivers and sockets activated in our configuration, excluding ones used for debugging (e.g., `/dev/gup_test`) or requiring specific hardware/architecture, as testing them would be meaningless or impractical. This filtering process is largely automated: debug drivers are easily identified by their `_test` suffix, and hardware-specific drivers can be filtered by focusing only on bootable modules.

Validation. We leverage two tools in Syzkaller, `syz-extract` and `syz-generate` to validate the generated specifications, which can detect many types of errors, including issues such as undefined types, wrong macro names, unmatched dependencies, and more.

5 Evaluation

We conduct an extensive evaluation on a workstation with 96 cores and 512 GB RAM, running Ubuntu 20.04.5 LTS. We selected the Linux kernel version 6.7 (`d2f51b`) as our target. Following prior work [25], we use the `allyesconfig` kernel configuration for specification generation, but use the `syzbot` [4] configuration from Google to build a bootable kernel for evaluation. We use the Syzkaller setting for fuzzing, with 4 QEMU instances, each utilizing 2 CPU cores. For baselines, we choose SyzDescribe [25], the state-of-the-art syscall specification generation approach, and existing Syzkaller [5] specifications, crafted by human experts.

5.1 Specification Generation for Missing

Under the **`allyesconfig`** option, KernelGPT scans *666 driver* operation and *85 socket* operation handlers. Out of the them, 278 and 81 are respectively **loaded** under the **`syzbot`** option. For these loaded driver and socket operation handlers, we generated *missing* descriptions, as presented in Table 1. After analysis, 75 driver and 66 socket operation handlers are missing one or more syscall descriptions (Column "# Incomplete"). Figure 7 presents a histogram where the x-axis represents the percentage of missing syscall specifications, and the y-axis shows the count of handlers at each percentage level. We note that Syzkaller does not have any description for many of these driver handlers (45 out of 75, or 60%). Plus, 22 socket handlers lack descriptions for over 80% of their syscalls. These findings underscore the insufficient specification of some drivers and sockets for effective testing, highlighting the need to synthesize additional specifications.

Table 1. Specifications for driver/socket handlers

			SyzDescribe	KernelGPT
	# Total	# Incomplete	# Valid	# Valid (Fixed)
Driver	278	75	20	70 (30)
Socket	81	66	N/A	57 (12)
Total	359	141	20	127 (42)

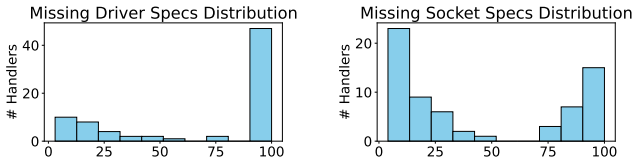

Figure 7. Missing specification distribution

Table 2. Newly generated syscall descriptions

	SyzDescribe		KernelGPT	
	# Syscalls	# Types	# Syscalls	# Types
Driver	146	168	288	170
Socket	N/A	N/A	244	124
Total	146	168	532	294

5.1.1 Statistics of New Specifications.

Among the 75 driver and 66 socket handlers with missing syscall descriptions, 40 and 45 are directly validated as correct (§ 3.1), and additional 30 and 12 are successfully repaired (§ 3.2). Hence, KernelGPT successfully generates specifications for 70 (93%) and 57 (86%) of the driver and socket handlers with missing syscalls, demonstrating the effectiveness of KernelGPT for specification generation and repair. By contrast, the state-of-the-art syscall specification generation approach, SyzDescribe [25], is limited to generating specifications for only 20 (27%) inadequately described driver handlers and cannot analyze socket handlers at all.

The descriptions generated by KernelGPT for these handlers includes 532 (13.6%) *new* syscalls, in addition to the 3903 existing syscalls described by Syzkaller, as shown in

Table 3. Overall effectiveness of KernelGPT (3 rep.)

	Cov	Unique Cov	Crash
Syzkaller	204,923	-	16.0
Syzkaller + SyzDescribe	201,634	14,585	13.7
Syzkaller + KernelGPT	209,673	20,472	17.7

Table 2. This also includes additional 294 descriptions for new types employed within these syscall specifications. By contrast, SyzDescribe has only 146 (3.7%) new syscall descriptions and 168 new type definitions for the drivers. Again, SyzDescribe lacks support for analyzing sockets, resulting in "N/A" entries in the corresponding table sections.

KernelGPT takes 4.7 hours to generate specifications for these 532 syscalls, more efficient than SyzDescribe, which requires 3.8 hours for just 146 syscalls. It processes approximately 5.56 million input tokens and generates 400,000 output tokens, with an average of 2,630 input and 189 output tokens per prompt. The total cost of $34 is negligible considering these specifications guide extensive fuzzing campaigns that typically run for days or weeks.

LLM-generated specifications offer unmatched readability compared to previous techniques. While existing methods often use random numbers for syscall names, file descriptor names, and struct field names, KernelGPT leverages LLM's ability to generate meaningful names, closely resembling expert-written specifications. For example, we have been requested by Syzkaller developers to upstream our generated specification for the CEC driver after we reported several bugs in the driver. Our generated specification, covering 12 syscalls and 10 structs/unions with 47 fields in total, was merged into Syzkaller with only one word changed manually [51]. In contrast, a previous attempt to merge a number of SyzDescribe generated specifications led to lengthy code review discussions and was not merged [50]. A quote from Syzkaller developers also highlights the importance of specification readability: *"Lots of automated descriptions that I saw are unreadable ... When you start digging they turn out to be bad in some way, but discovering that is extremely hard, it should be easy (e.g. literal constant names)"* [3].

5.1.2 Coverage Improvement by New Specifications. To show the effectiveness of the new specifications synthesized by KernelGPT for kernel fuzzing, we integrate them with the existing Syzkaller specifications, resulting in a combined suite (Syzkaller + KernelGPT). We conduct a 24-hour fuzzing session (192 CPU hours). For comparison, we also run the original Syzkaller and a combined suite of Syzkaller with the descriptions generated by SyzDescribe (Syzkaller + SyzDescribe), each under identical conditions and with three repetitions. The results are depicted in Table 3, where the Syzkaller + KernelGPT suite covers 4,750 and 8,039 more basic blocks than the original Syzkaller and the SyzDescribe integration, respectively. Additionally, KernelGPT adds 20,742 *unique* basic blocks to Syzkaller's coverage, in contrast to SyzDescribe's contribution of 14,585 additional unique blocks. These results underscore the contribution of KernelGPT-generated missing descriptions to enhancing coverage in kernel fuzzing.

5.1.3 Correctness of New Specifications. To assess the semantic correctness of KernelGPT-generated specifications, we manually examined the specifications for 45 drivers devoid of descriptions in Syzkaller (detailed in § 5.1), encompassing a total of 313 `IOCTL` syscall descriptions. Primarily, we focused on syscalls that KernelGPT overlooked, those with incorrect identifiers, and syscalls featuring erroneous types.

Regarding missing syscalls, we discovered that the majority of drivers (42/45, 93.3%) did not omit any syscall. For the three drivers with missing syscall descriptions, their actual syscall handling was delegated to other functions, sometimes even multiple times. This underscores the challenges LLMs face in analyzing indirect function calls. Moreover, we identified only 3 (0.9%) syscalls out of 2 (4.4%) drivers with incorrect identifier values. Upon closer inspection, we determined that modifications to identifier values, such as `if (DRM_IOCTL_NR(cmd) == DRM_COMMAND)`, which checks the modified identifier value, and `DRM_COMMAND` not being the true identifier value in this context, were responsible. Fortunately, LLMs only occasionally made mistakes in this regard, as evidenced by one driver with 16 such modified identifier values, yet only one of them was inferred incorrectly by the LLM. Lastly, only 9 syscalls out of 7 drivers exhibited incorrect types. Overall, these results show that KernelGPT can infer specifications with high accuracy and completeness.

5.1.4 Bug Detection by New Specifications. Table 4 shows the unknown kernel vulnerabilities detected using the newly generated specifications by KernelGPT. KernelGPT has detected 24 previously unknown bugs, with 21 confirmed by the kernel developers. 11 of them are assigned with CVE numbers, and 12 are already fixed. Notably, *none of them* can be detected by the default Syzkaller or SyzDescribe since they are only triggered by the new descriptions generated by KernelGPT, emphasizing the effectiveness of KernelGPT in revealing real-world kernel bugs. 17 bugs are detected from the drivers/sockets that have been loaded in the default `syzbot` configuration for an extended period but Syzkaller lacks specifications for them. Interestingly, the other 7 bugs were not revealed by Syzkaller because their specifications are incomplete. For example, Syzkaller's descriptions for the RDS socket [54] cover only the `recvmsg` syscall, omitting `sendto`. By generating the missing `sendto` specification, KernelGPT uncovered an array index out-of-bounds vulnerability in it, which was acknowledged with a CVE and patched by kernel developers. Next, we analyze and discuss two additional CVEs that are in the non-described drivers.

Table 4. New bugs detected by KernelGPT

Crash with new specs	New	Confirmed	Fixed	CVE	Syzkaller	SyzDescribe
kmalloc bug in ctl_ioctl	✓	✓	✓	CVE-2024-23851	×	×
kmalloc bug in dm_table_create	✓	✓	✓	CVE-2023-52429	×	×
KASAN: slab-use-after-free Read in cec_queue_msg_fh	✓	✓	✓	CVE-2024-23848	×	×
ODEBUG bug in cec_transmit_msg_fh	✓	✓	✓		×	×
WARNING in cec_data_cancel	✓	✓	✓		×	×
INFO: task hung in cec_claim_log_addrs	✓	✓			×	×
general protection fault in cec_transmit_done_ts	✓	✓	✓		×	×
kernel BUG in btrfs_get_root_ref	✓	✓	✓	CVE-2024-23850	×	×
general protection fault in btrfs_update_reloc_root	✓	✓			×	×
zero-size vmalloc in ubi_read_volume_table	✓	✓	✓	CVE-2024-25739	×	×
UBSAN: array-index-out-of-bounds in rds_cmsg_recv	✓	✓	✓	CVE-2024-23849	×	×
memory leak in ubi_attach	✓	✓		CVE-2024-25740	×	×
memory leak in posix_clock_open	✓	✓	✓	CVE-2024-26655	×	×
memory leak in __ip6_append_data	✓	✓			×	×
possible deadlock in dvb_demux_release	✓				×	×
INFO: task hung in __rq_qos_throttle	✓				×	×
WARNING in usb_ep_queue	✓	✓		CVE-2024-25741	×	×
memory leak in dvb_dmxdev_add_pid	✓	✓			×	×
memory leak in dvb_dvr_do_ioctl	✓				×	×
general protection fault in dvb_vb2_expbuf	✓	✓	✓	CVE-2024-50291	×	×
general protection fault in cleanup_mapped_device	✓	✓	✓	CVE-2024-50277	×	×
WARNING in vb2_core_reqbufs	✓	✓			×	×
BUG: corrupted list in vep_queue	✓	✓			×	×
divide error in uvc_queue_setup	✓	✓			×	×
Total	**24**	**21**	**12**	**11**	**0**	**0**

CVE-2024-23848. This vulnerability, titled *KASAN: slab-use-after-free Read in cec_queue_msg_fh*, is found within the CEC driver, for which Syzkaller lacks descriptions. It accesses a variable after it has been deallocated by `kfree(fh)`. The issue stems from the driver's failure to properly maintain a lock while releasing resources, leading to a Use-After-Free vulnerability. This bug has been rectified by kernel developers and has been assigned a CVE due to its exploitability.

CVE-2024-23851. This bug, *kmalloc bug in ctl_ioctl*, is detected by KernelGPT in the device mapper driver, which is also not described by Syzkaller. The root cause is that the driver neglects to check the allocation size for `kvmalloc`, leading to the possibility of allocating excessively large memory sizes. Specifically, the issue is associated with the `date_size` field in the `dm_ioctl` struct. This field plays a crucial role in allocating memory during the preparation of the data structure within `copy_param`. These elements are key in the process of allocating targets while executing `dm_table_create`. Notably, although SyzDescribe generates a specification for this driver, it incorporates an incorrect device filename, an erroneous command value, and imprecise types, thereby failing to detect this vulnerability. *Linus Torvalds confirmed this bug [1] in addition to providing detailed fixing suggestions since it required an in-depth understanding of the entire Linux codebase.* Given its potential for exploitation in DoS attacks, it has also been assigned a CVE.

5.2 Specification Generation for Existing

To further evaluate the quality of KernelGPT-generated specifications in terms of fuzzing, we apply it to generate specifications for the "existing" drivers and sockets described by our baselines, Syzkaller and SyzDescribe [25], the state-of-the-art specification generation techniques. We opt for all the 30 drivers used in the evaluation setting of SyzDescribe, as detailed in Table 6 of their paper [25]. For sockets, we compare against Syzkaller only. Regarding SyzDescribe, it cannot analyze and generate descriptions for sockets, attributed to the extensive implementation efforts required. We randomly selected 10 socket handlers using a seed value of 0, after arranging them in alphabetical order. We run each generated specification independently for 6 hours (48 CPU hours) with 3 repetitions to compare the coverage results. During these runs, we specifically enabled only the syscalls included in the specification for each driver or socket.

5.2.1 Device Drivers. Table 5 presents the results for drivers. Due to space constraints, we omit the average number of unique crashes for each driver in Table 5. In total, KernelGPT triggers 24.0 unique crashes, compared to 21.0 by Syzkaller and 20.7 by SyzDescribe. Two baseline drivers, `ashmem` and `fd#`, are no longer supported in Linux 6 ("N/A"). Notably, KernelGPT achieves the *highest* basic block coverage and crashes, surpassing the baselines by at least 18.0% and 17.6%. Moreover, KernelGPT performs the best on 20 of 28 (excluding the 2 invalid drivers, highlighted in bold),

Table 5. Comparison of driver specification generation with state-of-the-art solutions. *#Sys* is the number of syscalls described for the drivers. *Cov* represents the average coverage.

	Syzkaller		SyzDescribe		KernelGPT	
	#Sys	Cov	#Sys	Cov	#Sys	Cov
ashmem	N/A	-	N/A	-	N/A	-
btrfs-control	1	1523	5	**2848**	5	2786
capi20	13	2818	19	3011	14	**3138**
controlC#	22	4666	Err	-	15	**4703**
fd#	N/A	-	N/A	-	N/A	-
fuse	2	1719	2	2315	2	**2425**
hpet	1	1591	7	2289	7	**2493**
i2c-#	10	4168	10	4024	10	**4475**
kvm	118	10948	165	9444	71	**15605**
loop-control	4	7042	4	8211	4	**8537**
loop#	12	8498	12	**8519**	12	8518
mISDNtimer	3	**1992**	3	1965	3	1960
nbd#	11	4103	13	5311	12	**5475**
nvram	1	1618	3	2329	6	**2341**
ppp	24	5710	41	6102	34	**7509**
ptmx	49	**11598**	41	10870	30	11344
qat_adf_ctl	6	2788	6	2651	6	**2883**
rfkill	3	2117	4	**2388**	3	2301
rtc#	24	4458	33	4596	17	**5513**
sg#	39	**7412**	30	6414	43	7392
snapshot	13	3076	16	3260	15	**3470**
sr#	1	2882	68	3725	58	**5091**
timer	16	3328	Err	-	17	**3621**
udmabuf	4	2771	25	2115	4	**2921**
uinput	22	5470	24	4714	21	**6397**
usbmon#	9	3646	16	3806	9	**4332**
vhost-net	34	**3615**	25	3435	22	3541
vhost-vsock	3	2911	25	3448	22	**3803**
vmci	18	3760	26	4316	18	**4674**
vsock	1	1541	2	**1821**	2	1744
Total	464	117769	625*	113927	482	**138992**

whereas Syzkaller and SyzDescribe lead in only 4 and 4, respectively. This highlights the effectiveness of KernelGPT-generated specifications in enhancing fuzzing. For `kvm` driver [53], KernelGPT identifies two additional operation handlers, `kvm_vm_fops` and `kvm_vcpu_fops`, as dependencies. This leads to a coverage increase of *42.5%* and *65.2%* compared to baselines.

While SyzDescribe has the largest set of specifications, it repeatedly describes the same `ioctl` syscall using different types, which is atypical. An `ioctl` command accepts only a single type in most scenarios. Excluding duplicates, KernelGPT describes more distinct syscalls (482) than SyzDescribe (464). Additionally, SyzDescribe incorrectly inferred device names for `controlC#` and `timer`, preventing coverage.

5.2.2 Sockets. Table 6 presents the results for sockets. We observe that KernelGPT can cover *18.6%* more basic blocks than our baseline Syzkaller, demonstrating the effectiveness of our generated specifications for sockets. Notably, KernelGPT describes significantly more syscalls than Syzkaller,

Table 6. Comparison on socket specification generation.

	Syzkaller			KernelGPT		
	# Sys.	Cov	Crash	# Sys.	Cov	Crash
caif_stream	4	8947	0.7	6	**11902**	0.7
l2tp_ip6	38	**18350**	0.7	99	18080	0.7
llc_ui	10	7648	0.3	24	**16437**	0.0
mptcp	22	10480	1.3	70	**13942**	0.7
packet	22	**22082**	0.3	25	21363	0.3
phonet_dgram	7	11426	1.0	12	**15202**	0.7
pppol2tp	10	**18789**	0.3	14	12379	0.7
rds	11	13693	0.3	19	**17462**	1.0
rfcomm_sock	22	7263	1.0	16	**10893**	0.7
sco_sock	20	11349	1.0	19	**16527**	0.7
Total	166	130027	7.0	304	**154187**	6.0

showcasing KernelGPT's capability in syscall discovery. This is also because Syzkaller often uses a single syscall with various command values, while KernelGPT generates distinct syscalls for each command value. For example, KernelGPT synthesizes 99 descriptions for the `l2tp_ip6` socket handler, compared to Syzkaller's 38. This is because the single Syzkaller syscall of this socket, `getsockopt$inet6_int`, uses `flags[inet6_option_types_int]` as the command value *list*, encompassing 45 unique syscall identifier values.

5.2.3 Ablation Study. We perform a comprehensive ablation study to evaluate how KernelGPT's behavior would be affected if different components of it were disabled or modified. Due to resource limitations, we selected only the first 10 *valid* drivers from Table 5.

Iterative Multi-Stage Generation. We experimented with a new setup where all function code related to syscalls was combined into a single prompt and the LLM generated the specification in one step, deviating from our iterative multi-stage generation process. This simplified approach, however, resulted in a noticeable decline in the quality of the generated specifications, especially for complex drivers like `kvm` and `loop#`. For example, iterative multi-stage prompting infers 71 syscalls and 28 types for `kvm`, while an all-in-one prompting approach infers only 42 syscalls and 11 types, resulting in a substantial decline in coverage (15,605 versus 5,457). Overall, iterative multi-stage prompting can infer 1.28X more syscalls and 2.37X more types, leading to a 1.39X improvement in coverage for these 10 drivers.

LLM Choice. We examined the performance of KernelGPT when utilizing other LLMs, specifically GPT-3.5 and GPT-4o. Our observations revealed that employing GPT-3.5 significantly reduced the number of described syscalls (85 versus 143), leading to a 21% decrease in coverage compared to our default choice, GPT-4. Conversely, when using GPT-4o, KernelGPT was able to infer a similar number of syscalls (144 versus 143), and the coverage results were comparable to our default choice, GPT-4 (55,771 versus 54,640). We

would conclude that the syscall specification inference task necessitates a sufficiently powerful model like GPT-4 and GPT-4o.

6 Related Work

6.1 Kernel Fuzzing

SyzGen [14] infers syscall specifications but targets binary-only macOS drivers, leveraging symbolic execution to recover the data types and syscall traces to find dependencies. Moonshine [42] collects and distills syscall traces to generate a seed pool for Syzkaller. SyzVegas [55] leverages reinforcement learning to dynamically improve seed and task selection. HEALER [48] infers syscall dependencies by observing coverage changes with different combinations. Plus, SyzDirect [52] applies directed grey-box fuzzing for Syzkaller by incorporating distance information. PrIntFuzz [34] uses static analysis to extract driver information and generate their simulators for fuzzing. Focusing on synthesizing specifications, KernelGPT is orthogonal to the above techniques and could be combined to improve Syzkaller collectively.

6.2 Learning-Based Fuzzing

Machine learning approaches to input generation [16, 22, 32, 43, 45] explored using sequence-to-sequence models to learn program syntax and generation patterns. For instance, Learn&Fuzz [22] leveraged sequence-to-sequence models for grammar-based fuzzing by learning from sample inputs. DeepFuzz [32] extended this approach to generate C programs for compiler testing. DeepSmith [16] also demonstrated the potential of learning program patterns from a large corpus. However, these approaches faced significant limitations: they required extensive domain-specific training data, exhibited lower throughput compared to traditional fuzzers, and struggled to leverage existing fuzzing infrastructure. Our initial explorations show that even using more powerful LLMs for direct input generation in kernel fuzzing performed notably worse than basic Syzkaller.

Recent advancements have shown that LLMs [13, 31, 37, 38, 44] excel in a variety of natural language processing [10] and programming tasks [11, 58, 60, 62]. Their proficiency in diverse tasks is attributed to the extensive training on vast datasets, *e.g.,* GPT4 [38] is pre-trained using trillions of text tokens from the entire Internet. As a result, LLMs can be employed in various tasks simply by following instructions [8, 37, 41], eliminating the need for specialized training. Recently, a growing body of research has focused on leveraging LLMs for software testing, covering both unit test generation [30, 36, 46, 66] and fuzzing [17, 26, 35, 59, 63]. For example, TitanFuzz [17] pioneered the application of modern LLMs for both generation-based [33, 65] and mutation-based [18, 29] fuzzing, while Fuzz4All [59] further demonstrated that the multilingual potentials of LLMs can be utilized to serve as a universal fuzzer for a wide range of software systems.

KernelGPT takes a fundamentally different approach from both traditional ML-based and recent LLM-based fuzzing techniques. Instead of directly generating test inputs, we integrate LLMs with mature fuzzing frameworks by synthesizing their *components* (*i.e.,* input generators). This strategy leverages the expertise and resources invested in well-developed fuzzing tools while capitalizing on LLMs' capabilities. To the best of our knowledge, KernelGPT is the first work to successfully apply LLMs to kernel fuzzing, demonstrating state-of-the-art performance in generating high-quality syscall specifications.

7 Conclusion

In this paper, we propose KernelGPT, the first approach to synthesizing syscall specifications automatically via LLMs for enhanced kernel fuzzing. It employs an iterative method to autonomously deduce syscall specifications and further repair them using validation feedback. Experimental results show that KernelGPT helps improve Syzkaller's coverage and can detect 24 previously unknown bugs through the newly generated specifications, with 11 CVE assignments and 12 fixed. Additionally, a number of specifications inferred by KernelGPT are already merged into Syzkaller repository, following a request from its development team. To our knowledge, this is the first automated approach to leveraging LLMs for kernel fuzzing. It could open up numerous possibilities for future research in this critical application domain.

Acknowledgments

We are grateful to the anonymous reviewers and our shepherd, Youngjin Kwon, for their valuable feedback that helped improve this paper. This work was partially supported by NSF grant CCF-2131943 and Kwai Inc. Chenyuan Yang was partially supported by Boeing for research on Linux kernel testing. We also thank Ziqi Zhang for his helpful suggestions on the manuscript.

References

[1] Email thread with Linus. https://lore.kernel.org/all/XXX@XXX.com/T/#xxx.
[2] GPT4 Turbo. https://platform.openai.com/docs/models/gpt-4-and-gpt-4-turbo.
[3] sys/linux: automatic syscall interface extraction. https://github.com/google/syzkaller/issues/590.
[4] Syzbot. https://syzkaller.appspot.com/upstream/.
[5] Syzkaller. https://github.com/google/syzkaller/.
[6] syzlang. https://github.com/google/syzkaller/blob/master/docs/syscall_descriptions_syntax.md.
[7] The LLVM Compiler Infrastructure. https://llvm.org.

[8] Yejin Bang, Samuel Cahyawijaya, Nayeon Lee, Wenliang Dai, Dan Su, Bryan Wilie, Holy Lovenia, Ziwei Ji, Tiezheng Yu, Willy Chung, et al. A multitask, multilingual, multimodal evaluation of chatgpt on reasoning, hallucination, and interactivity. *arXiv preprint arXiv:2302.04023*, 2023.

[9] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[10] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[11] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, et al. Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv preprint arXiv:2303.12712*, 2023.

[12] Joseph Bursey, Ardalan Amiri Sani, and Zhiyun Qian. Syzretrospector: A large-scale retrospective study of syzbot, 2024.

[13] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[14] Weiteng Chen, Yu Wang, Zheng Zhang, and Zhiyun Qian. Syzgen: Automated generation of syscall specification of closed-source macos drivers. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 749–763, New York, NY, USA, 2021. Association for Computing Machinery.

[15] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. Difuze: Interface aware fuzzing for kernel drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138, 2017.

[16] Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. Compiler fuzzing through deep learning. In *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*, pages 95–105, 2018.

[17] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, 2023.

[18] Alastair F Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. Automated testing of graphics shader compilers. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–29, 2017.

[19] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. *SIGOPS Oper. Syst. Rev.*, 35(5):57–72, oct 2001.

[20] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[21] Marius Fleischer, Dipanjan Das, Priyanka Bose, Weiheng Bai, Kangjie Lu, Mathias Payer, Christopher Kruegel, and Giovanni Vigna. {ACTOR}:{Action-Guided} kernel fuzzing. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 5003–5020, 2023.

[22] Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn&fuzz: Machine learning for input fuzzing. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 50–59. IEEE, 2017.

[23] NCC Group. Triforce Linux Syscall Fuzzer. https://github.com/nccgroup/TriforceLinuxSyscallFuzzer.

[24] HyungSeok Han and Sang Kil Cha. Imf: Inferred model-based fuzzer. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2345–2358, New York, NY, USA, 2017. Association for Computing Machinery.

[25] Yu Hao, Guoren Li, Xiaochen Zou, Weiteng Chen, Shitong Zhu, Zhiyun Qian, and Ardalan Amiri Sani. Syzdescribe: Principled, automated, static generation of syscall descriptions for kernel drivers. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 3262–3278. IEEE Computer Society, 2023.

[26] Jie Hu, Qian Zhang, and Heng Yin. Augmenting greybox fuzzing with generative ai. *arXiv preprint arXiv:2306.06782*, 2023.

[27] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. Survey of hallucination in natural language generation. *ACM Computing Surveys*, 55(12):1–38, 2023.

[28] Dave Jones. Trinity. https://github.com/kernelslacker/trinity.

[29] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. *ACM Sigplan Notices*, 49(6):216–226, 2014.

[30] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K Lahiri, and Siddhartha Sen. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *International conference on software engineering (ICSE)*, 2023.

[31] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023.

[32] Xiao Liu, Xiaoting Li, Rupesh Prajapati, and Dinghao Wu. Deepfuzz: Automatic generation of syntax valid c programs for fuzz testing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 1044–1051, 2019.

[33] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. Random testing for c and c++ compilers with yarpgen. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–25, 2020.

[34] Zheyu Ma, Bodong Zhao, Letu Ren, Zheming Li, Siqi Ma, Xiapu Luo, and Chao Zhang. Printfuzz: fuzzing linux drivers via automated virtual device simulation. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2022, page 404–416, New York, NY, USA, 2022. Association for Computing Machinery.

[35] Ruijie Meng, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury. Large language model guided protocol fuzzing. In *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS)*, 2024.

[36] Pengyu Nie, Rahul Banerjee, Junyi Jessy Li, Raymond J Mooney, and Milos Gligoric. Learning deep semantics for test completion. *arXiv preprint arXiv:2302.10166*, 2023.

[37] OpenAI. Chatgpt. 2023. https://openai.com/blog/chatgpt.

[38] OpenAI. Gpt-4 technical report, 2023.

[39] Oracle. Kernel-Fuzzing. https://github.com/oracle/kernel-fuzzing.

[40] Palash B. Oswal. *Improving Linux Kernel Fuzzing*. PhD thesis, 2023.

[41] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744, 2022.

[42] Shankara Pailoor, Andrew Aday, and Suman Jana. {MoonShine}: Optimizing {OS} fuzzer seed selection with trace distillation. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 729–743, 2018.

[43] Mohit Rajpal, William Blum, and Rishabh Singh. Not all bytes are equal: Neural byte sieve for fuzzing. *ArXiv*, abs/1711.04596, 2017.

[44] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

[45] Martin Sablotny, Bjørn Sand Jensen, and Chris W. Johnson. Recurrent neural networks for fuzz testing web browsers. In *International Conference on Information Security and Cryptology*, 2018.

[46] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. Adaptive test generation using a large language model. *arXiv preprint arXiv:2302.06527*, 2023.

[47] Hao Sun, Yuheng Shen, Jianzhong Liu, Yiru Xu, and Yu Jiang. KSG: Augmenting kernel fuzzing with system call specification generation. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 351–366, Carlsbad, CA, July 2022. USENIX Association.

[48] Hao Sun, Yuheng Shen, Cong Wang, Jianzhong Liu, Yu Jiang, Ting Chen, and Aiguo Cui. Healer: Relation learning guided kernel fuzzing. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 344–358, 2021.

[49] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[50] Syzkaller Project. sys/linux: syz-describe: auto generate syzlang. https://github.com/google/syzkaller/pull/3143, 2022.

[51] Syzkaller Project. Merged specifications in syzkaller. https://github.com/google/syzkaller/pull/xxxx, 2024.

[52] Xin Tan, Yuan Zhang, Jiadong Lu, Xin Xiong, Zhuang Liu, and Min Yang. Syzdirect: Directed greybox fuzzing for linux kernel. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 1630–1644, New York, NY, USA, 2023. Association for Computing Machinery.

[53] The Linux Kernel documentation. Kvm. https://docs.kernel.org/virt/kvm/index.html, 2024.

[54] The Linux Kernel documentation. Rds. https://docs.kernel.org/networking/rds.html, 2024.

[55] Daimeng Wang, Zheng Zhang, Hang Zhang, Zhiyun Qian, Srikanth V Krishnamurthy, and Nael Abu-Ghazaleh. {SyzVegas}: Beating kernel fuzzing odds with reinforcement learning. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2741–2758, 2021.

[56] Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. Magicoder: Source code is all you need. *arXiv preprint arXiv:2312.02120*, 2023.

[57] Wikipedia contributors. Device mapper — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Device_mapper&oldid=1146533552, 2023. [Online; accessed 28-December-2023].

[58] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489*, 2024.

[59] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. Universal fuzzing via large language models. *arXiv preprint arXiv:2308.04748*, 2023.

[60] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. Automated program repair in the era of large pre-trained language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1482–1494, 2023.

[61] Chunqiu Steven Xia and Lingming Zhang. Keep the conversation going: Fixing 162 out of 337 bugs for $0.42 each using chatgpt. *arXiv preprint arXiv:2304.00385*, 2023.

[62] Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pages 1–10, 2022.

[63] Chenyuan Yang, Yinlin Deng, Runyu Lu, Jiayi Yao, Jiawei Liu, Reyhaneh Jabbarvand, and Lingming Zhang. Whitefox: White-box compiler fuzzing empowered by large language models. 8(OOPSLA2), October 2024.

[64] Chenyuan Yang, Yinlin Deng, Jiayi Yao, Yuxing Tu, Hanchi Li, and Lingming Zhang. Fuzzing automatic differentiation in deep-learning libraries. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1174–1186. IEEE, 2023.

[65] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 283–294, 2011.

[66] Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. No more manual tests? evaluating and improving chatgpt for unit test generation. *arXiv preprint arXiv:2305.04207*, 2023.

[67] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. The fuzzing book, 2019.