KernelGPT: Enhanced Kernel Fuzzing via Large Language Models

Chenyuan Yang
UIUC
cy54@illinois.edu

Zijie Zhao
UIUC
zijie4@illinois.edu

Lingming Zhang
UIUC
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. Syzkaller, one of the most widely studied kernel fuzzers, aims to generate valid syscall sequences based on predefined specifications written in syzlang, a domain-specific language for defining syscalls, their arguments, and the relationships between them. While there has been existing work trying to automate Syzkaller specification generation, this still 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 inferring Syzkaller specifications via Large Language Models (LLMs) for enhanced kernel fuzzing. Our basic 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 all the necessary specification components, and further leverages the validation feedback to repair/refine the initial specifications. Our preliminary results demonstrate that KernelGPT can help Syzkaller achieve higher coverage and find multiple previously unknown bugs. Moreover, we also received a request from the Syzkaller team to upstream specifications inferred by KernelGPT.

1 Introduction

Operating system kernels are among the most critical systems globally, as all other types of systems rely on and operate on them. Vulnerabilities in a kernel, such as crashes or out-of-bounds writes, can be maliciously exploited, potentially causing substantial harm to all users of that operating system kernel. To ensure the correctness and security of these fundamental systems, fuzzing (or fuzz testing) [42, 55] has been employed for decades. Such techniques automatically generate a vast number of system calls (syscalls) as test inputs, intending to detect potential kernel bugs.

Among various kernel fuzzing techniques [19, 23, 34], Syzkaller [3] stands out as one of the most popular and effective, having identified over 4.9K bugs that were later fixed by developers [2]. Concurrently, numerous research efforts have focused on enhancing Syzkaller, targeting areas such as seed generation [35, 37], seed selection [45], guided mutation [18, 41], and syscall specification generation [11, 12, 21, 40]. Among these advancements, the syscall descriptions written in syzlang [4] stand out as one of the most crucial components, significantly contributing to the effectiveness of Syzkaller and allowing it to cover more kernel modules. These descriptions specify the types of syscalls and their inter-dependencies, enabling the generation of more valid syscall sequences that probe deeper into the kernel code logic. However, crafting syscall specifications is predominantly a manual process, requiring 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 [12] and SyzDescribe [21] employ static code analysis to identify device driver syscall handlers and infer their corresponding descriptions. Figure 1 illustrates the workflow of static analysis-based techniques in the upper part. Initially, experts manually define rules to translate device source code into descriptions, drawing upon their own understanding of the kernel codebase and existing Syzkaller examples. These rules are then laboriously hard-coded into the static analysis tool, a process that is both challenging and time-consuming. The accuracy and effectiveness of the generated syscall descriptions depend heavily on the comprehensiveness of these mapping rules, a task that 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.

Take, for instance, Figure 2a and Figure 2b, which illustrates the source code of two `struct` variables, associated

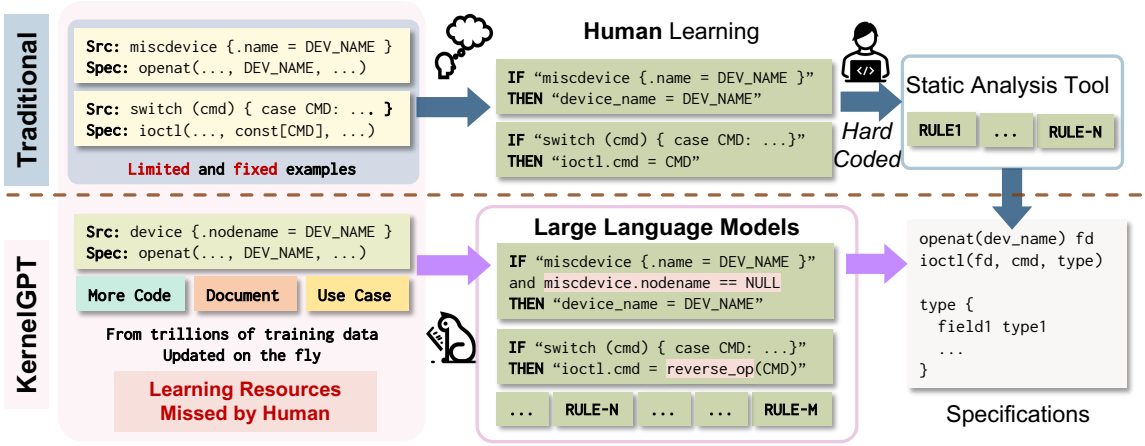

Figure 1: Workflows of syscall specification inference based on static analysis and LLMs

with the device mapper driver [48], 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 [21], typically rely on the `.name` field 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 `.nodename` field, a legitimate but rare use case, leading to an incorrect inference by SyzDescribe. Moreover, SyzDescribe also fails to accurately analyze the command value for `ioctl`, the syscall 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 the user-provided value. 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.

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) [10, 17, 28, 32, 33, 38, 47] are pre-trained on vast datasets, including kernel codebases, documentations, and real-world syscall use cases. This extensive training means that LLMs have been exposed to and may have learned far more information related to syscall description inference than average human experts. Consequently, they are potentially more adept at analyzing the source code of drivers, even those implemented in various styles, and in generating high-quality 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 infer more accurately with 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 two previously unknown bugs in the kernel.

Building on the insight discussed earlier, we introduce KernelGPT, the first approach to automated syscall specification generation by using Large Language Models (LLMs), focusing on kernel drivers. The key idea of KernelGPT is to employ LLMs for automating and enhancing the rule inference process, aimed at synthesizing high-quality driver descriptions from their source code. Initially, KernelGPT uses LLMs to deduce the device name and its initialization descriptions based on the device operation handlers and their references. To recover the command values, argument types, and type definitions for the driver, KernelGPT iteratively applies LLMs to analyze the associated source code, which LLMs themselves indicate from previous analysis steps. Finally, 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 [13, 49, 52], 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 all specification components for kernel drivers with a novel iterative strategy and further repair the descriptions with the validation feed-

```
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`

back. While the basic idea of KernelGPT is general, we leverage state-of-the-art GPT4 as the analysis LLM to infer the descriptions based on the code extracted by the kernel code parser.

- We evaluate the ability of KernelGPT in generating specifications for previously undescribed drivers to detect bugs and for existing described drivers to compare against state-of-the-art baselines, SyzDescribe and Syzkaller. Our preliminary results show that the descriptions generated for the existing drivers by KernelGPT can achieve 21.3% more coverage than baselines. By now, KernelGPT has detected 8 crashes in the new drivers, with 7 previously unknown. Notably, we have received a request from the Syzkaller team to upstream our generated specifications.

2 Background and Related Work

2.1 Kernel and Device Drivers

Kernel. An operating system kernel provides userspace applications with key functionalities, such as virtual memory, file system, networking, accessing devices, and more. To protect the safety of all applications and users, interactions between userspace and kernel are confined to a well-defined system call (syscall) interface *e.g.,* the POSIX standard. Kernel bugs and crashes that can be triggered by userspace applications through syscalls pose a significant risk since they affect all applications using the kernel and an exploit could bypass all security policies enforced by the kernel. Therefore, detecting kernel bugs through the syscall interface is critical.

Device drivers. In Linux, devices are abstracted as files, typically located in the `/dev` directory, and can be accessed through the same syscall mechanism. Device drivers register their syscall handlers with the kernel upon initialization. The kernel will then dispatch syscalls from userspace to the appropriate driver handler.

With the same example in Figure 2, the driver initially constructs the `struct file_operations` in Figure 2a, which maps syscalls to specific driver handlers. Fields such as `.open` and `.unlocked_ioctl` store function pointers to driver handlers `dm_open` and `dm_ctl_ioctl`, respectively. The driver then creates the `struct miscdevice`, incorporating the `struct file_operations` and naming the device file in the `.nodename` field. Finally, the driver calls `misc_register(&_dm_misc)` to register the driver with the kernel. When an application attempts to open the file denoted by the `.nodename`, equivalent to `/dev/mapper/control`, the kernel invokes the registered handler `dm_open` and associate its returned open file descriptor (`fd`) with the device mapper driver. The kernel will dispatch future syscalls with the `fd` to the corresponding registered handler. For instance, `ioctl(fd, ...)` will trigger `dm_ctl_ioctl`.

Driver developers often implement standard control logic in handlers like `.open`, `.close`, and `.release`. However, many drivers also require unique control logic that has no similar counterpart in the syscall interface, *e.g.,* the device mapper driver requires an operation to get a list of all the `dm` device names (Figure 2d). For such driver-specific operations, developers commonly use the generic `ioctl(int fd, unsigned long request, void *argp)`[1] syscall as a dispatcher to invoke the corresponding driver function. The first parameter `fd` is an open file descriptor of the device file. The second parameter `request` serves as a command identifier to select the operation to perform. The third parameter `argp` is cast to driver-specific data types to carry information in and out. For instance, to get the list of `dm` device names, an application would first construct the driver-specific structure `struct dm_ioctl data = ...;` and open the device file by `int fd = open("/dev/mapper/control");` Then the device names can be retrieved by `ioctl(fd, DM_LIST_DEVICES, &data);` Since each driver may require a unique data type associated with a specific command identifier, such specialized usage turns `ioctl` into thousands of syscalls whose interface is largely not well-documented or standardized.

[1]The actual definition `ioctl(int fd, unsigned long request, ...)` is a variadic function but typically only one untyped pointer is used.

2.2 Kernel Fuzzing

To address the complexity and continual evolution of operating system kernels, a variety of sophisticated methods have been employed for decades to identify kernel bugs or vulnerabilities [11, 12, 16, 19–21, 23, 37, 41, 43, 45]. Fuzz testing, or fuzzing, is among the most effective techniques. Kernel fuzzers generate syscalls and execute them on the target kernel, typically with sanitizers enabled, until a crash occurs. Syzkaller [3] is the state-of-the-art coverage-guided kernel fuzzer, having identified and aided the rectification of thousands of kernel vulnerabilities [2]. Syzkaller respects the intricate data structures used in kernel and inter-dependencies among syscalls. Syzkaller uses a domain-specific language, syzlang [4], to define syscall specifications (or descriptions), guiding the generation of test cases. Syscall specifications enable Syzkaller to generate semantically valid syscall sequences to reach deep code paths within the kernel.

Syzkaller specification. Expressing parameter types and dependencies for a syscall without specifying any concrete parameter value is impossible. Again take `ioctl` as an example, the underlying type of the untyped pointer depends on the command identifier value, which in turn depends on the concrete file name used by the previous call to `open`. To define such value-dependent types and dependencies, Syzkaller allows defining multiple "instances" of the same syscall, where each instance can have concrete values for some or all of its parameters. Each instance is treated as a unique syscall and thus can have its own specification and dependencies defined.

Figure 3 shows the specification of three syscalls for the MSM driver (some parameters and names are omitted or shortened for simplicity). The syscall `openat$msm` is an instance of the syscall `openat`, where `msm` is a custom but unique name to distinguish syscall instances. The specification defines a concrete parameter value `"/dev/msm"` for the `openat$msm` - the name of the MSM device. For parameters that can be dynamically created by Syzkaller, the specification outlines their types, *e.g.,* the `arg` parameter for `ioctl$NEW` and `ioctl$CLOSE` are pointers to a structure `drm_msm_submitqueue` and an integer `msm_submitqueue_id`, respectively.

A special type in syzlang, **resource**, is used to express dependencies between syscalls. A resource has to be generated by a call before being used as input to other calls. The specification in Figure 3 introduces two resources, `fd_msm` which represents an open file descriptor, and `msm_submitqueue_id`, which represents a queue ID used internally by MSM driver. The resource `fd_msm` is returned by `openat$msm` and then used as an input argument by `ioctl$NEW` and `ioctl$CLOSE`. As a result, Syzkaller will only put `ioctl$NEW` and `ioctl$CLOSE` after `openat$msm`. More fine-grained dependency is also supported, such as specifying dependencies on one field in a structure. For instance, the `inout` annotation for the `arg` parameter of `ioctl$NEW` indicates that the structure `drm_msm_submitqueue` is used both as input and output. In `drm_msm_submitqueue`, the

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

Figure 3: Specification for the MSM driver in syzlang

field `msm_submitqueue_id` has the annotation `out`, meaning it is the output. Since `ioctl$CLOSE` takes `msm_submitqueue_id` as input, `ioctl$CLOSE` can only be generated after a `ioctl$NEW` has populated the `msm_submitqueue_id`. Without the specification, it would be challenging, if not impossible, for the fuzzer to correctly construct the expected string and the `drm_msm_submitqueue` structure, and generate a sequence adhering to the dependencies. However, with the specification in place, Syzkaller can drastically cut down the search space and focus on fuzzing the internal fields of `drm_msm_submitqueue`.

Specification generation. Despite their effectiveness, specifications are typically manually written by Syzkaller and kernel developers, requiring deep expertise in the kernel and the specific kernel module. Thus, existing Syzkaller specifications only cover a subset of syscalls, especially lacking specifications for device drivers. Existing specifications can also become out-of-date as kernel evolves [21]. Automating specification generation is clearly desired but faces several key challenges. One is to extract expected parameter values and type definitions. Another is to infer dependencies among syscalls and encode the dependencies in parameters. Failing to address these challenges would lead to inaccurate specifications, diminishing the effectiveness of a fuzzing campaign.

Several techniques for specification generation have been proposed, attempting to address the above challenges. KSG [40] dynamically finds syscall handler structures by opening existing device files and probing the kernel to find the accessed data structure. It then collects the type and range information through symbolic execution. DIFUZE [12] and SyzDescribe [21] both conduct static analysis on kernel source code, identifying common implementation patterns to generate specifications. DIFUZE finds syscall handlers from a list of data structures used by common device registration functions. SyzDescribe starts by finding the initialization functions for kernel modules and tracing down to find the function pointers to the syscall handlers. Both DIFUZE and SyzDescribe follow certain programming patterns to extract device names and command identifiers, *e.g.,* a switch case inside the handler is likely invoking the corresponding sub-handlers based on the command value. Existing specification generation approaches mainly rely on hard-coding the human-summarized patterns in analysis tools, to translate the source code implementation to the syscall specifications. In contrast, KernelGPT leverages the potential of LLMs to automate and

improve the learning of description inference rules, resulting in better performance.

Other kernel fuzzing work. SyzGen [11] also generates syscall specifications but targets binary-only macOS drivers, leveraging symbolic execution to recover the data types required by syscalls and syscall traces to find syscall dependencies. Moonshine [37] collects and distills syscall traces to generate a seed pool for Syzkaller. SyzVegas [45] leverages reinforcement learning to dynamically improve seed and task selection. HEALER [41] infers syscall dependencies by observing coverage changes with different syscall combinations. Plus, SyzDirect [43] applies directed grey-box fuzzing for Syzkaller by incorporating distance information as feedback. ThunderKaller [24] improves Syzkaller performance by skipping block calls, debloating coverage collection, and sanitizers. Focusing on generating specifications from source code, KernelGPT is orthogonal to the above techniques and could be combined with them to improve Syzkaller collectively.

2.3 LLMs for Testing

Recent advancements have shown that Large Language Models (LLMs) [10, 28, 32, 33, 38] excel in a variety of natural language processing [7] and programming tasks [9, 51], including code comprehension and document generation. Their proficiency in diverse tasks is attributed to the extensive training on vast datasets, *e.g.,*GPT4 [33] 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 [6, 32, 36], eliminating the need for specialized training.

A growing body of research has focused on leveraging LLMs for testing, such as unit test generation [26, 31, 39, 54], fuzzing [13, 14, 22, 30, 49, 52] and static analysis [27]. TitanFuzz [13] is the pioneering work in applying modern LLMs for both generation-based [29, 53] and mutation-based [15, 25, 46] fuzzing, specifically targeting machine learning systems. Fuzz4All [49] further demonstrates that the multilingual potentials of LLMs can be utilized to serve as a universal fuzzer for a wide range of software systems. More recently, WhiteFox [52] leverages the implementation source code of compiler optimizations as guidance for LLMs to perform white-box compiler fuzzing. Existing works in utilizing LLM in testing mainly focus on using LLMs to create the test inputs for the systems under test directly, *e.g.,* generating C programs for testing C compilers. By contrast, KernelGPT integrates the potential of LLMs with established fuzzing frameworks by synthesizing their components, which is complimentary to existing LLM-based fuzzing techniques and offers additional benefits by leveraging the expertise and resources invested in well-developed fuzzing tools. Furthermore, to our knowledge, KernelGPT is the first work to leverage LLMs for kernel fuzzing and demonstrate their ability to generate high-quality syscall specifications.

3 Approach

Figure 4 presents the overview of KernelGPT, which utilizes the code extractor and analysis LLM to automatically generate driver specifications for enhancing kernel fuzzing. KernelGPT takes the kernel codebase and located device operation handlers as input and operates through three automated phases: Driver Detection ❶, Specification Generation ❷, and Specification Validation and Repair ❸.

Initially, KernelGPT identifies drivers by employing LLMs to infer the device names and their initialization specifications, using the details of the operation handlers and their usage (§ 3.1 ❶). Subsequently, KernelGPT determines the command values, argument types, and type definitions for describing the `ioctl` handlers of the devices. 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 (§ 3.2 ❷). Lastly, KernelGPT validates the generated specifications. If errors are found, it attempts to repair the descriptions by consulting the LLMs with the error messages (§ 3.3 ❸).

We assume that the locations for the device operation handlers are known for KernelGPT since they can be identified by simple yet general patterns. Therefore, we use a specialized code parser implemented with LLVM [5] to search the kernel codebases to pinpoint instances where the device operation handler structs are initialized with `ioctl` handler functions. More specifically, we search for initialization instances of the `ioctl` or `unlocked_ioctl` fields within the operation handler structs. For instance, the device mapper driver shown in Figure 2a initiates the `unlocked_ioctl` field in `_ctl_fops` structure by using `dm_ctl_ioctl` function. We label `dm_ctl_ioctl` as the `ioctl` handler and `_ctl_fops` as the device operation handler. Notably, the focus of KernelGPT is on the inference from the source code to descriptions, rather than locating the device operation or `ioctl` handlers. Hence, we employ a straightforward pattern-searching method to find the device operation and `ioctl` handlers.

3.1 Driver Detection

To deduce the device name of the driver, our approach begins with locating references to the device operation handler, utilizing the kernel code extractor. Following this, we employ the usage information to guide the LLMs in inferring the device name. To enhance the accuracy, we apply few-shot in-context learning techniques [8], crafting specific prompts that aid the model in better understanding the nuances of device name determination.

Figure 5 illustrates the example prompt used for inferring device names. It includes the instruction, the source code of

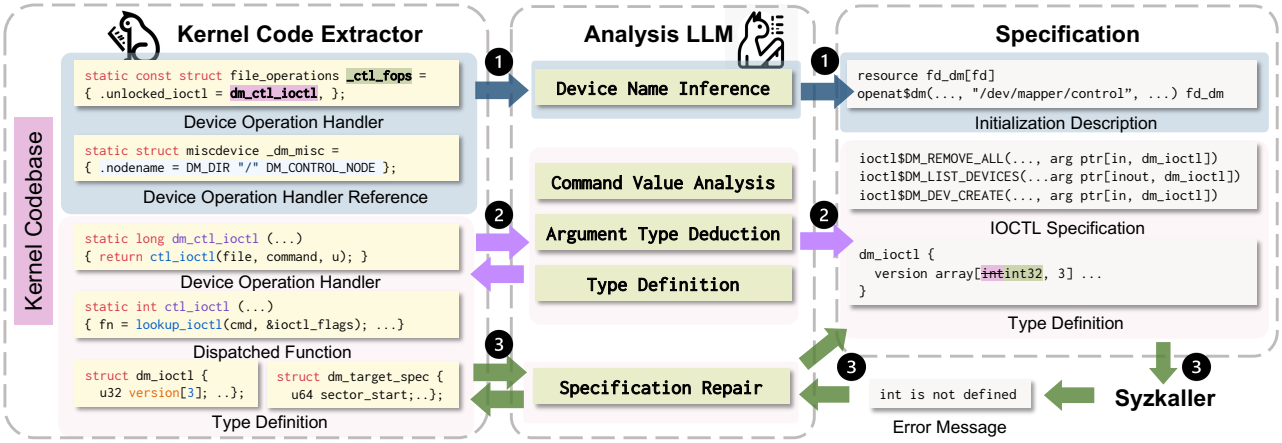

Figure 4: Overview of KernelGPT

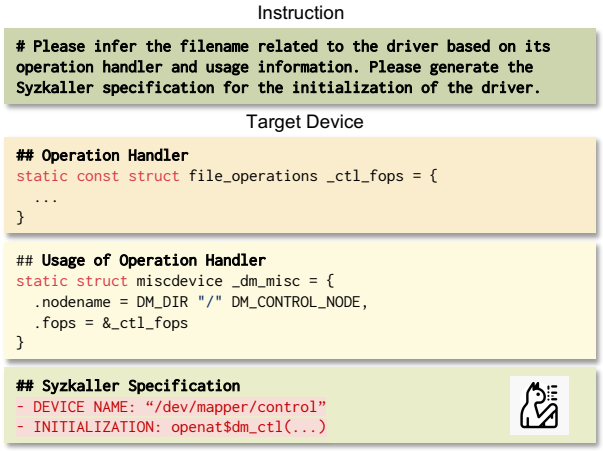

Figure 5: Prompt for device name inference

Algorithm 1: Iterative Analysis by LLMs

```
1  Function Analyze(relatedCode, usageInfo, step):
2      if step > MAX_ITER then
3          return NULL
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      successResult, unknown ← QueryLLM(prompt)
8      for (funcs, types, usageInfo) ∈ unknown do
9          # Extract the code for all missing functions and types
10         relatedCode ← ExtractCode(funcs, types)
11         # Iteratively analyze for the next iteration
12         res ← Analyze(relatedCode, usageInfo, step + 1)
13         # Update with the result analyzed for the unknown
14         Update(successResult, res)
15     return successResult
```

the operation handler, and its usage, concluding with the specification that is to be generated by LLMs. We anticipate that the LLMs will not only determine the device name (`DEVICE NAME` in Figure 5) but also analyze the initialization descriptions associated with the device (`INITIALIZATION`). Typically, the initialization is described with the syscall `openat` or `syz_open_dev` for most drivers. Referring back to the device mapper driver, the operation handler `_ctl_fops` is utilized during the initialization of `struct miscdevice _dm_misc`, as shown in Figure 5. Through the analysis of this source code usage, LLMs can determine the correct device name, in this case, `mapper/control`, based on the `nodename` field within the `miscdevice` struct. Since LLMs have assimilated such knowledge during their pre-training phase, they are capable of inferring the correct device name and initialization specification with the appropriate reference information.

3.2 Specification Generation

In this phase, we generate the descriptions of `ioctl` for drivers by leveraging LLMs to analyze their implementation source code. To enhance the performance of LLMs, we segment the process into three distinct stages: deducing the command value, identifying the argument type, and defining the type. This structured approach enables LLMs to concentrate on one specific aspect at each stage, thereby improving efficiency and focus. Similar to the device name inference process, we utilize in-context few-shot prompting [8] at each stage of the specification generation. By providing relevant examples, we aim to enhance LLMs' comprehension of the task and familiarize them with the expected output format.

We develop a novel iterative method to utilize and improve the code analysis capabilities of LLMs, which is motivated by two primary factors. Firstly, despite state-of-the-art LLMs such as GPT4 supporting a context size of 128K [1], it remains impractical to provide LLMs with the entirety of the

source code related to a driver. Secondly, our goal is to deduce the command value and argument type for the `ioctl` handler, and not all the code or helper functions within the `ioctl` handler are pertinent to this objective. Recognizing that some parts may not contribute to our aim, we have devised a strategy where LLMs analyze the kernel code recursively, as shown in Algorithm 1. Firstly, we generate a few-shot example prompt with the source code related to the target and its usage informaton (Line 5). Then, LLMs are queried to infer the descriptions and pinpoint any functions or types that are missing yet essential for inference, marking with the keyword `UNKNOWN` for identification (Line 7). For each unknown candidate, we extract the source code for all its missing functions and types (Line 10), which is supplied to LLMs in the subsequent step for further analysis (Line 12). The analysis concludes and outputs the results either when no unknown elements are detected or when the iteration count exceeds `MAX_ITER` (Line 3). Importantly, the entire analysis process is fully automated, steered by the unknown information provided by LLMs.

3.2.1 Command Value

We use an iterative prompting strategy for analyzing command values, as depicted in Figure 6. In addition to the instruction, this prompt includes the source code of functions relevant to the inference task. This encompasses the `ioctl` handler function and any associated helper functions pertinent to command value analysis. The expected output from LLMs is the set of successfully inferred command values. If the logic for checking command 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 `UNKNOWN` field). Furthermore, code snippets that utilize the command value variables are also included. Should LLMs identify any unknown command 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` field of the previous step serves as a reference for guiding subsequent steps.

Figure 6 presents the first two steps of command value analysis 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 the source code, LLMs are unable to deduce any command values and designate `ctl_ioctl` as the absent function. Subsequently, 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 command value, `DM_VERSION`, while other values related to the function `lookup_ioctl` remain undetermined. Thus, LLMs report `DM_VERSION` as the inferred value and `lookup_ioctl` as the missing function, necessitating further analysis in the next step.

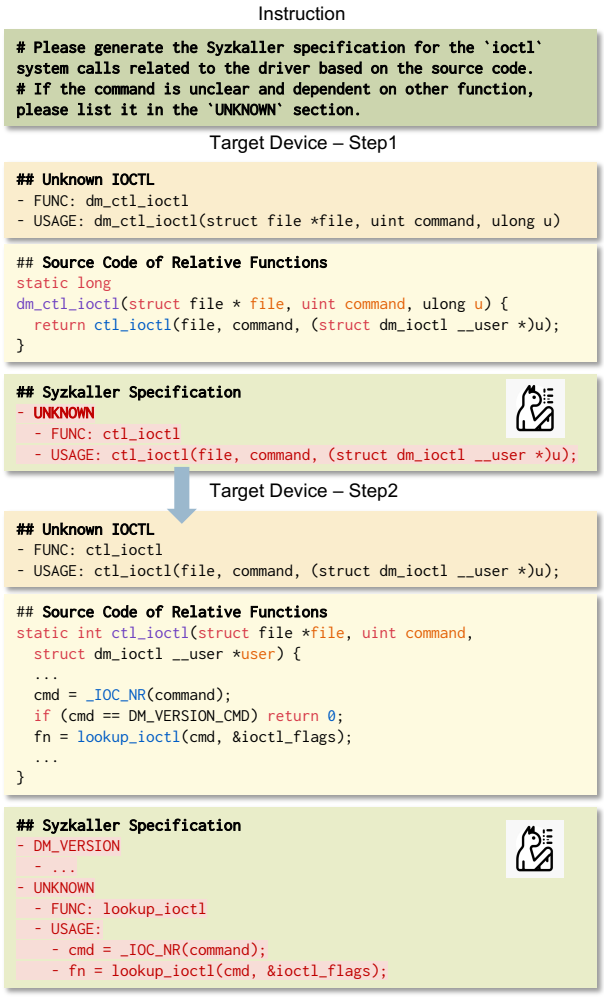

Figure 6: Prompt for command value analysis

3.2.2 Argument Type

After determining the command value, the subsequent stage involves analyzing the argument type, as depicted in the prompt shown in Figure 7. The initial information for this inference, derived from the command value inference stage, includes relevant functions and code snippets demonstrating the argument's usage. KernelGPT then extracts the source code for these related functions and presents them to LLMs, which are tasked with identifying the argument type. If the code logic that determines the argument type is delegated to another function, the type will still be marked as `UNKNOWN`. In such instances, KernelGPT continues its analysis, leveraging the new information provided by LLMs.

Figure 7 shows a practical example of argument inference for the `DM_REMOVE_ALL` command value in the device mapper driver. In the preceding step, LLMs identify the function associated with this argument as `remove_all`, along with its usage code. Consequently, KernelGPT supplies LLMs with the source code of `remove_all` to guide the argument type

```
## IOCTL
- DM_REMOVE_ALL
  - ARG: UNKNOWN
  - FUNC: remove_all
  - ARG_NAME_IN_USAGE: param
  - USAGE: r = remove_all(file, param, input_param_size);
```

```
## Source Code of Relative Functions
static int remove_all(struct file *filp, struct dm_ioctl *param,
size_t param_size) {
  dm_hash_remove_all(true,!!(param->flags & DM_DEFERRED_REMOVE));
  param->data_size = 0;
  return 0;
}
```

```
## Syzkaller Specification
- DM_REMOVE_ALL
  - ARG: ptr[in, dm_ioctl]
- TYPES
  - dm_ioctl
```

Figure 7: Prompt for argument type inference

```
## Source Code
typedef struct _PhysDevAddr_struct {
  DWORD TargetId:24;
  DWORD Bus:6;
  DWORD Mode:2;
  SCSI3Addr_struct Target[2]; /* 2 level target device addr */
} PhysDevAddr_struct
```

```
## Syzkaller Specification
- TYPES
  - PhysDevAddr_struct: { TargetId int32:24 ... }
  - SCSI3Addr_struct: UNKNOWN
```

Figure 8: Prompt for type definition generation

inference. Analyzing its signature, LLMs deduce that the argument should be a pointer to the struct `dm_ioctl`. Additionally, LLMs place the struct `dm_ioctl` in the `TYPES` field, setting it up for type definition analysis in the following stage.

3.2.3 Type Definition

After identifying the argument types, KernelGPT proceeds to generate descriptions for these types. Figure 8 displays the prompt used in this stage. Essentially, KernelGPT retrieves the definition source code of the type from the Linux kernel codebase. This source code is then presented to LLMs, tasked with creating the corresponding Syzkaller description. In cases with a reference to a nested type within the type definition, LLMs are instructed to mark it as `UNKNOWN` in their output. These marked types are then further analyzed in subsequent steps.

The type definition inference example depicted in Figure 8 showcases a specific case: one field of `PhysDevAddr_struct` is an array composed of the struct type `SCSI3Addr_struct`. As the source code for this newly referenced struct, `SCSI3Addr_struct`, is not included in the provided information, LLMs are expected to identify it as `UNKNOWN`. This designation indicates that `SCSI3Addr_struct` requires further analysis in the subsequent step.

Instruction

```
# Please repair the Syzkaller description based on the error
message and provided source code.
```

Target Wrong Description

```
## Wrong Syzkaller Description
vfio_pci_hot_reset_info {
  count int32
  devices array[vfio_pci_dependent_device, count]
}
```

```
## Error Message
count is unsupported on all arches (typo?)
```

```
## Source Code
...
```

```
## Correct Syzkaller Specification
vfio_pci_hot_reset_info {
  count len[devices, int32]
  devices ptr[inout, array[vfio_pci_dependent_device]]
}
```

Figure 9: Prompt for specification repair

3.3 Specification Validation and Repair

In this phase, inspired by recent work on LLM-based program repair [50], our goal is to validate the specifications generated by KernelGPT and automatically repair any specifications that are invalid. This step is crucial, as LLMs may occasionally make mistakes during the description generation process. To address this, we utilize the Syzkaller tool, `syz-extract`, which can assess the validity specifications. It is capable of detecting grammar errors in the specifications, including issues like undefined variables and types. Initially, KernelGPT uses the error messages from `syz-extract` to pinpoint inaccuracies in specific descriptions, effectively matching each error message to its corresponding description. Subsequently, for those descriptions identified with errors, KernelGPT queries LLMs for correction, guided by few-shot examples. This process, depicted in Figure 9, 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.

For example, as illustrated in Figure 9, the type description for `vfio_pci_hot_reset_info` is initially incorrect. This is due to syzlang's requirement for the length in `array[type, length]` to be a constant, whereas the description uses a variable-length array. The correct format for a variable-length array in syzlang is simply `array[type]`. By analyzing the error message "count is unsupported on all arches" generated by `syz-extract`, LLMs repair this issue by redefining `devices` as a variable-length array and specifying `count` as a type `len[devices]`, thereby aligning the description with the requirements of syzlang.

4 Implementation

Source code extractor. We implement the Linux kernel source code extractor by using LLVM toolchain [5]. This tool parses the kernel codebase to identify the device operation handlers by pattern mapping for preparing the input for KernelGPT. It then extracts the `ioctl` handlers used within the operation handlers, along with their corresponding reference locations. It also compiles all definitions of `function`, `struct`, `union`, and `enum` found within the kernel. These definitions are then utilized as guidance for LLMs in the specification generation and repair processes, provided when LLMs indicate their necessity.

Analysis LLM. While our approach is general and, consequently, independent of the specific LLMs used, our tool, KernelGPT, is constructed atop the state-of-the-art LLM, GPT4 [33]. At each step, we utilize the OpenAI APIs to query GPT4, with a low-temperature parameter of `0.1`. For `MAX_ITER`, the stopping criteria for analysis, we set it as 5 by default.

Few-shot prompting. For inferring the device name and command value, we employ a 3-shot prompting technique to accommodate the context size limitations of LLMs. This is necessary due to the often lengthy function source codes involved in these two steps, which necessitate a restriction on the number of few-shot examples. In contrast, for other stages such as argument type inference and type definition analysis, we opt for 6-shot prompting. This allows us to fully exploit the available context size. During the repair phase, given the variety of potential errors and the typically concise nature of the necessary repair information and error messages, we choose a 9-shot prompting strategy. This approach is designed to provide LLMs with a broader understanding of the repair process.

Driver selection. While our approach is general, in this preliminary evaluation, we only concentrate on generating descriptions for drivers or handlers that are not described in Syzkaller. This focus is crucial because drivers lacking specifications are typically not tested thoroughly. As such, these drivers warrant more attention compared to those that already have specifications. In this process, after analyzing the device name, we first verify whether Syzkaller already has descriptions for the device. If there are no existing descriptions in Syzkaller for a particular device, we then proceed to generate them. Furthermore, our primary focus is on regular drivers, including character and block devices. We leave the specification generation for network and USB devices for future work.

5 Evaluation

We investigate the following research questions in our experiments to evaluate our approach:

- **RQ1:** What is the number and quality of the specifications for undescribed kernel drivers generated by KernelGPT?
- **RQ2:** What is the quality of the syscall descriptions generated by KernelGPT compared to the baselines?
- **RQ3:** Can specifications generated by KernelGPT detect real-world bugs in the kernel?

We conduct a preliminary evaluation to answer these research questions, on a workstation equipped with 64 cores and 256 GB RAM, running Ubuntu 20.04.5 LTS. We selected the Linux kernel version 6.7 as our target, which was released on November 5, 2023, identified by the hash `d2f51b`. Furthermore, for both specification generation and evaluation, we use the `syzbot` [2] configuration for the Linux kernel. This choice reflects the decision made by Google in their fuzzing of the Linux kernel through QEMU. Our fuzzing configuration follows the default Syzkaller setting with 4 QEMU instances, each utilizing 2 CPU cores. To ensure a fair comparison and lessen the effects of crash reproduction, we have disabled the `reproduce` feature while collecting and comparing coverage results. To demonstrate the quality of specifications generated by KernelGPT, we choose SyzDescribe [21], the state-of-the-art syscall specification generation approach, and existing Syzkaller [3] specifications, crafted by human experts, as our baselines.

5.1 Overall Results

Table 1: Specification generation for the undescribed drivers

Generation		Evaluation and Repair		Executaion
# Total	# Success	# Valid	# Valid after repair	# Executable
50	39/50	24/39	32/39	17/32

By using the kernel code extractor with general patterns, we detect 132 `ioctl` handlers under the `syzbot` kernel configuration, excluding the network and USB driver handlers. Among these devices, 50 (37.9%) are identified as lacking descriptions in Syzkaller, based on their device names deduced by KernelGPT. We then generate descriptions for these undescribed drivers, successfully inferring 39 of them, as presented in Table 1.

The failure to infer descriptions for the remaining devices can primarily be attributed to several reasons. Firstly, the complex logic of their code makes it challenging for LLMs to understand, especially when the source code of the `ioctl` function is overly lengthy, hindering GPT4's comprehension. Additionally, when the source code to be analyzed exceeds the context limit of GPT4, it directly fails to infer correctly.

Out of the 39 specifications successfully generated by KernelGPT, 24 are directly validated as correct, and an additional 8 are successfully repaired by KernelGPT, enabling them to pass the validation check. However, KernelGPT is unable to repair the remaining 7 specifications. For the 32 valid specifications, we run Syzkaller with these validated specifications,

Table 2: Results of the executable undescribed drivers

`ioctl` Handler	# Descriptions	Cov	Unique Cov
btrfs_control_ioctl	4	2719	20
cec_ioctl	12	3643	402
ctrl_cdev_ioctl	3	6190	596
dm_ctl_ioctl	19	5692	837
dvb_demux_ioctl	10	4534	632
dvb_dvr_ioctl	2	4115	172
dvb_frontend_ioctl	17	3508	196
hci_uart_tty_ioctl	6	9201	183
hpet_ioctl	7	2665	27
joydev_ioctl	7	4798	87
kcov_ioctl	4	4188	152
mtdchar_unlocked_ioctl	26	5168	233
nvram_misc_ioctl	3	2206	8
pty_unix98_ioctl	2	7799	173
serport_ldisc_ioctl	2	11894	2040
sr_block_ioctl	3	5791	74
uart_ioctl	2	6254	836
Total	129	90365	6668

and 17 of them can be executed. One potential reason for the non-executable devices is that they necessitate complex initialization steps, which cannot be defined in specifications (*e.g.,* helper functions like `syz_open_dev`) or cannot be accurately inferred by GPT4. Another contributing factor could be that these devices are not enabled in the `syzbot` kernel configuration.

We run each newly detected driver separately for 8 hours with our default setting. The results are shown in Table 2, where Column "# Descriptions" presents the number of system call descriptions for each device, and Column "Cov" is the number of lines covered when fuzzing each device exclusively with our generated specification for 8 hours. Additionally, we consider the unique coverage, shown in the "Unique Cov" column. This metric reflects the unique coverage obtained when compared to the standard Syzkaller setup, which activates all 3,912 system calls under identical conditions. In this standard setup, 143,838 lines are covered. The specifications for the previously unaddressed drivers, generated by KernelGPT, add 129 (3.3%) more descriptions and contribute to covering an additional 6,668 (5%) unique lines. These figures underscore the significance of including descriptions for the previously unaddressed drivers and demonstrate the effectiveness of the specifications produced by KernelGPT.

In addition to running the specifications for each driver separately, we integrate all the specifications for the new drivers with the original Syzkaller specifications. This means that in a single run, we execute both the new specifications generated by KernelGPT and the existing specifications in Syzkaller. Given the extensive number of approximately four thousand system calls, we extend the fuzzing duration to 24 hours for this combined run. Subsequently, we compared the coverage results from this combined run with those obtained from running only the existing Syzkaller specifications under the same setting. Integrating specifications synthesized by KernelGPT for the undescribed drivers has proven to be effective, resulting in 28.6% (28 vs. 36) more crashes being triggered compared to using only the existing Syzkaller descriptions. In terms of code coverage, the results are more modest, largely due to the preliminary nature and scale of our experiment. With KernelGPT introducing a relatively small number of new syscall descriptions (129) compared to the total available (3912), the increase in code coverage post-integration is modest, with an additional 1.5K lines covered. Nonetheless, even this limited-scale experiment highlights the significant potential for using KernelGPT to infer specifications for a larger number of syscalls, which could lead to much more effective kernel fuzzing in future applications.

5.2 Comparison with Baselines

To evaluate the quality of the specifications generated by KernelGPT, we choose 10 "existing" devices described by our baselines, Syzkaller and SyzDescribe [21], which is the state-of-the-art specification generation technique.We apply KernelGPT to generate the descriptions for the selected devices. Specifically, we opt for the first 10 drivers listed in the evaluation setting of SyzDescribe, as detailed in Table 6 of their paper [21]. Each of these drivers is described in both baseline methods. Subsequently, we run each generated specification independently for 8 hours to compare the coverage results by only enabling the syscalls under the specification for each driver. During these runs, we specifically enabled only the syscalls included in the specification for each driver.

Table 3 presents the results of the specifications generated by KernelGPT, SyzDescribe, and Syzkaller, detailing the number of syscalls, defined types, and the line coverage after 8 hours of fuzzing. Notably, KernelGPT achieves the highest line coverage, surpassing the baselines by more than 21.3%. This underscores both the superior quality of specifications produced by KernelGPT and its effectiveness in enhancing fuzzing performance. Moreover, KernelGPT defines the highest number of types for the drivers, demonstrating its robust capabilities in type analysis. For the baseline SyzDescribe, two of the tested specifications failed to produce coverage. This issue stemmed from incorrect device names inferred by SyzDescribe for the drivers `controlC#` and `timer`. While both are actually located in the `/dev/snd` directory, SyzDescribe incorrectly identifies their location as `/dev`. As a result, the devices could not be opened, leading to no coverage. Furthermore, the overall comparison results demonstrate that KernelGPT can even surpass Syzkaller on the described drivers, highlighting the promising future of applying KernelGPT on all possible drivers.

5.3 Kernel Bug Detection

Table 4 presents the bugs detected by KernelGPT in the undescribed drivers. By now, KernelGPT has detected 8 unique bugs, with 7 previously unknown. Notably, all of them cannot be detected by the default Syzkaller.

Table 3: Comparison of specification generation with state-of-the-art solutions

Device Name	`ioctl` Handler Names	KernelGPT # Syscalls	KernelGPT # Types	KernelGPT Cov	SyzDescribe # Syscalls	SyzDescribe # Types	SyzDescribe Cov	Syzkaller # Syscalls	Syzkaller # Types	Syzkaller Cov
controlC#	snd_ctl_ioctl	18	18	**5477**	21	10	N/A	25	8	4057
loop-control	loop_control_ioctl	4	0	7550	4	0	**7809**	4	2	7638
loop#	lo_ioctl	12	3	6482	12	3	**6986**	12	2	5465
rfkill	rfkill_fop_ioctl	2	0	**2572**	4	0	2327	3	1	2331
rtc#	rtc_dev_ioctl	16	3	**5174**	33	4	4840	23	3	4260
sg#	sd_ioctl	15	6	**7747**	30	1	6659	40	9	6576
snapshot	snapshot_ioctl	9	1	**3616**	15	1	3455	15	1	2887
sr#	sr_block_ioctl	50	17	**6538**	68	7	4940	1	0	2791
usbmon#	usbdev_ioctl, mon_bin_ioctl	39	13	**3843**	70	19	3794	38	6	3410
timer	snd_timer_user_ioctl	17	9	3153	21	9	N/A	16	9	**3597**
	Total	182	70	**52152**	278	54	40810	177	41	43012

Table 4: Bugs detected by KernelGPT

Crash in New Drivers	All	New
WARNING: kmalloc bug in ctl_ioctl	1	1
WARNING: kmalloc bug in dm_table_create	1	1
KASAN: slab-use-after-free Read in cec_queue_msg_fh	1	1
WARNING: ODEBUG bug in cec_transmit_msg_fh	1	1
WARNING in cec_data_cancel	1	1
INFO: task hung in cec_claim_log_addrs	1	1
general protection fault in cec_transmit_done_ts	1	1
general protection fault in vidtv_mux_stop_thread	1	0
Total	8	7

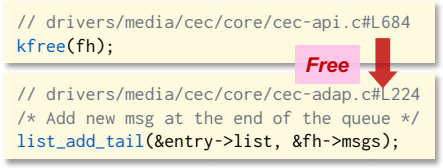

Figure 10: slab-use-after-free Read in cec_queue_msg_fh

Bug analysis. We analyze the root cause of the 7 new bugs, all of which are detected in two newly described drivers, device mapper [48] and CEC [44].

The first two bugs, *kmalloc bug in ctl_ioctl* and *kmalloc bug in dm_table_create*, are detected in the device mapper driver. Their root causes are different but similar: the driver neglects to check the allocation size for `kvmalloc`, leading to the possibility of allocating excessively large memory sizes. Specifically, the issue in the former crash 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`. For the latter crash, the problem centers around the `DM_TABLE_LOAD_CMD` command value and the `target_count` field. 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 these two bugs. The first bug has already been confirmed by the kernel developer.

The remaining newly discovered bugs were all detected in the CEC driver, a standardized kernel interface for HDMI CEC hardware. Figure 10 includes code snippets associated with the bug *KASAN: slab-use-after-free Read in cec_queue_msg_fh*, where the CEC driver reads from a variable that has been freed by `kfree(fh)`. The bug *ODEBUG bug in cec_transmit_msg_fh* occurs when the driver tries to free an active object with `kfree(data)`. The *WARNING in cec_data_cancel* is triggered by an internal check within the CEC driver, which expects the variable to be in either a current or pending state. The issue *INFO: task hung in cec_claim_log_addrs* happens when the kernel hangs as the CEC device waits for the completion of a configuration task. Lastly, the *general protection fault in cec_transmit_done_ts* occurs when the CEC device attempts to dereference a non-canonical address, resulting in a kernel crash.

It is noteworthy that the specifications generated by KernelGPT have proven helpful in kernel bug detection. Consequently, we have received a request from the Syzkaller team to integrate our generated specifications into its upstream repository.

6 Conclusion

In this paper, we propose KernelGPT, the first approach to generating syscall specifications automatically via LLMs for enhanced kernel fuzzing. It employs an iterative method to autonomously deduce all necessary components of a specification and further repairs these specifications using validation feedback. Preliminary results indicate that KernelGPT helps to improve Syzkaller's coverage and detects 7 previously unknown bugs in the newly described drivers. Additionally, the Syzkaller team has expressed interest in incorporating specifications inferred by KernelGPT into their main repository. 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, encompassing applications of LLMs in seed generation, selection, and mutation for Syzkaller-style fuzzing, as well as direct syscall generation and more.

References

[1] GPT4 Turbo. https://platform.openai.com/docs/models/gpt-4-and-gpt-4-turbo.

[2] Syzbot. https://syzkaller.appspot.com/upstream/.

[3] Syzkaller. https://github.com/google/syzkaller/.

[4] syzlang. https://github.com/google/syzkaller/blob/master/docs/syscall_descriptions_syntax.md.

[5] The LLVM Compiler Infrastructure. https://llvm.org.

[6] BANG, Y., CAHYAWIJAYA, S., LEE, N., DAI, W., SU, D., WILIE, B., LOVENIA, H., JI, Z., YU, T., CHUNG, W., ET AL. A multitask, multilingual, multimodal evaluation of chatgpt on reasoning, hallucination, and interactivity. *arXiv preprint arXiv:2302.04023* (2023).

[7] BROWN, T., MANN, B., RYDER, N., SUBBIAH, M., KAPLAN, J. D., DHARIWAL, P., NEELAKANTAN, A., SHYAM, P., SASTRY, G., ASKELL, A., ET AL. Language models are few-shot learners. *Advances in neural information processing systems 33* (2020), 1877–1901.

[8] BROWN, T., MANN, B., RYDER, N., SUBBIAH, M., KAPLAN, J. D., DHARIWAL, P., NEELAKANTAN, A., SHYAM, P., SASTRY, G., ASKELL, A., ET AL. Language models are few-shot learners. *Advances in neural information processing systems 33* (2020), 1877–1901.

[9] BUBECK, S., CHANDRASEKARAN, V., ELDAN, R., GEHRKE, J., HORVITZ, E., KAMAR, E., LEE, P., LEE, Y. T., LI, Y., LUNDBERG, S., ET AL. Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv preprint arXiv:2303.12712* (2023).

[10] CHEN, M., TWOREK, J., JUN, H., YUAN, Q., PINTO, H. P. D. O., KAPLAN, J., EDWARDS, H., BURDA, Y., JOSEPH, N., BROCKMAN, G., ET AL. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[11] CHEN, W., WANG, Y., ZHANG, Z., AND QIAN, Z. Syzgen: Automated generation of syscall specification of closed-source macos drivers. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2021), CCS '21, Association for Computing Machinery, p. 749–763.

[12] CORINA, J., MACHIRY, A., SALLS, C., SHOSHITAISHVILI, Y., HAO, S., KRUEGEL, C., AND VIGNA, G. Difuze: Interface aware fuzzing for kernel drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017), pp. 2123–2138.

[13] DENG, Y., XIA, C. S., PENG, H., YANG, C., AND ZHANG, L. 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* (2023), ISSTA 2023.

[14] DENG, Y., XIA, C. S., YANG, C., ZHANG, S. D., YANG, S., AND ZHANG, L. Large language models are edge-case fuzzers: Testing deep learning libraries via fuzzgpt. *arXiv preprint arXiv:2304.02014* (2023).

[15] DONALDSON, A. F., EVRARD, H., LASCU, A., AND THOMSON, P. Automated testing of graphics shader compilers. *Proceedings of the ACM on Programming Languages 1*, OOPSLA (2017), 1–29.

[16] ENGLER, D., CHEN, D. Y., HALLEM, S., CHOU, A., AND CHELF, B. Bugs as deviant behavior: A general approach to inferring errors in systems code. *SIGOPS Oper. Syst. Rev. 35*, 5 (oct 2001), 57–72.

[17] FENG, Z., GUO, D., TANG, D., DUAN, N., FENG, X., GONG, M., SHOU, L., QIN, B., LIU, T., JIANG, D., ET AL. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[18] FLEISCHER, M., DAS, D., BOSE, P., BAI, W., LU, K., PAYER, M., KRUEGEL, C., AND VIGNA, G. {ACTOR}:{Action-Guided} kernel fuzzing. In *32nd USENIX Security Symposium (USENIX Security 23)* (2023), pp. 5003–5020.

[19] GROUP., N. Triforce Linux Syscall Fuzzer. https://github.com/nccgroup/TriforceLinuxSyscallFuzzer.

[20] HAN, H., AND CHA, S. K. Imf: Inferred model-based fuzzer. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2017), CCS '17, Association for Computing Machinery, p. 2345–2358.

[21] HAO, Y., LI, G., ZOU, X., CHEN, W., ZHU, S., QIAN, Z., AND SANI, A. A. Syzdescribe: Principled, automated, static generation of syscall descriptions for kernel drivers. In *2023 IEEE Symposium on Security and Privacy (SP)* (2023), IEEE Computer Society, pp. 3262–3278.

[22] HU, J., ZHANG, Q., AND YIN, H. Augmenting greybox fuzzing with generative ai. *arXiv preprint arXiv:2306.06782* (2023).

[23] JONES, D. Trinity. https://github.com/kernelslacker/trinity.

[24] LAN, Y., JIN, D., WANG, Z., TAN, W., MA, Z., AND ZHANG, C. Thunderkaller: Profiling and improving the performance of syzkaller. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2023), pp. 1567–1578.

[25] LE, V., AFSHARI, M., AND SU, Z. Compiler validation via equivalence modulo inputs. *ACM Sigplan Notices 49*, 6 (2014), 216–226.

[26] LEMIEUX, C., INALA, J. P., LAHIRI, S. K., AND SEN, S. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *International conference on software engineering (ICSE)* (2023).

[27] LI, H., HAO, Y., ZHAI, Y., AND QIAN, Z. Assisting static analysis with large language models: A chatgpt experiment. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2023), pp. 2107–2111.

[28] LI, R., ALLAL, L. B., ZI, Y., MUENNIGHOFF, N., KOCETKOV, D., MOU, C., MARONE, M., AKIKI, C., LI, J., CHIM, J., ET AL. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023).

[29] LIVINSKII, V., BABOKIN, D., AND REGEHR, J. Random testing for c and c++ compilers with yarpgen. *Proceedings of the ACM on Programming Languages 4*, OOPSLA (2020), 1–25.

[30] MENG, R., MIRCHEV, M., BÖHME, M., AND ROYCHOUDHURY, A. Large language model guided protocol fuzzing. In *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS)* (2024).

[31] NIE, P., BANERJEE, R., LI, J. J., MOONEY, R. J., AND GLIGORIC, M. Learning deep semantics for test completion. *arXiv preprint arXiv:2302.10166* (2023).

[32] OPENAI. Chatgpt. https://openai.com/blog/chatgpt.

[33] OPENAI. Gpt-4 technical report, 2023.

[34] ORACLE. Kernel-Fuzzing. https://github.com/oracle/kernel-fuzzing.

[35] OSWAL, P. B. *Improving Linux Kernel Fuzzing*. PhD thesis, 2023.

[36] OUYANG, L., WU, J., JIANG, X., ALMEIDA, D., WAINWRIGHT, C., MISHKIN, P., ZHANG, C., AGARWAL, S., SLAMA, K., RAY, A., ET AL. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems 35* (2022), 27730–27744.

[37] PAILOOR, S., ADAY, A., AND JANA, S. {MoonShine}: Optimizing {OS} fuzzer seed selection with trace distillation. In *27th USENIX Security Symposium (USENIX Security 18)* (2018), pp. 729–743.

[38] ROZIERE, B., GEHRING, J., GLOECKLE, F., SOOTLA, S., GAT, I., TAN, X. E., ADI, Y., LIU, J., REMEZ, T., RAPIN, J., ET AL. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).

[39] SCHÄFER, M., NADI, S., EGHBALI, A., AND TIP, F. Adaptive test generation using a large language model. *arXiv preprint arXiv:2302.06527* (2023).

[40] SUN, H., SHEN, Y., LIU, J., XU, Y., AND JIANG, Y. KSG: Augmenting kernel fuzzing with system call specification generation. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)* (Carlsbad, CA, July 2022), USENIX Association, pp. 351–366.

[41] SUN, H., SHEN, Y., WANG, C., LIU, J., JIANG, Y., CHEN, T., AND CUI, A. Healer: Relation learning guided kernel fuzzing. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (2021), pp. 344–358.

[42] SUTTON, M., GREENE, A., AND AMINI, P. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[43] TAN, X., ZHANG, Y., LU, J., XIONG, X., LIU, Z., AND YANG, M. Syzdirect: Directed greybox fuzzing for linux kernel. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2023), CCS '23, Association for Computing Machinery, p. 1630–1644.

[44] THE LINUX KERNEL DOCUMENTATION. Hdmi cec. https://docs.kernel.org/admin-guide/media/cec.html, 2023.

[45] WANG, D., ZHANG, Z., ZHANG, H., QIAN, Z., KRISHNAMURTHY, S. V., AND ABU-GHAZALEH, N. {SyzVegas}: Beating kernel fuzzing odds with reinforcement learning. In *30th USENIX Security Symposium (USENIX Security 21)* (2021), pp. 2741–2758.

[46] WEI, A., DENG, Y., YANG, C., AND ZHANG, L. Free lunch for testing: Fuzzing deep-learning libraries from open source. In *Proceedings of the 44th International Conference on Software Engineering* (2022), pp. 995–1007.

[47] WEI, Y., WANG, Z., LIU, J., DING, Y., AND ZHANG, L. Magicoder: Source code is all you need. *arXiv preprint arXiv:2312.02120* (2023).

[48] 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].

[49] XIA, C. S., PALTENGHI, M., TIAN, J. L., PRADEL, M., AND ZHANG, L. Universal fuzzing via large language models. *arXiv preprint arXiv:2308.04748* (2023).

[50] XIA, C. S., AND ZHANG, L. Keep the conversation going: Fixing 162 out of 337 bugs for $0.42 each using chatgpt. *arXiv preprint arXiv:2304.00385* (2023).

[51] XU, F. F., ALON, U., NEUBIG, G., AND HELLENDOORN, V. J. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming* (2022), pp. 1–10.

[52] YANG, C., DENG, Y., LU, R., YAO, J., LIU, J., JABBARVAND, R., AND ZHANG, L. White-box compiler fuzzing empowered by large language models. *arXiv preprint arXiv:2310.15991* (2023).

[53] YANG, X., CHEN, Y., EIDE, E., AND REGEHR, J. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation* (2011), pp. 283–294.

[54] YUAN, Z., LOU, Y., LIU, M., DING, S., WANG, K., CHEN, Y., AND PENG, X. No more manual tests? evaluating and improving chatgpt for unit test generation. *arXiv preprint arXiv:2305.04207* (2023).

[55] ZELLER, A., GOPINATH, R., BÖHME, M., FRASER, G., AND HOLLER, C. The fuzzing book, 2019.