ECG: Augmenting Embedded Operating System Fuzzing via LLM-based Corpus Generation

Qiang Zhang, Yuheng Shen, Jianzhong Liu, Yiru Xu, Heyuan Shi, Yu Jiang and Wanli Chang✉

***Abstract*—Embedded Operating Systems power much of our critical infrastructure but are, in general, much less tested for bugs than general-purpose operating systems. Fuzzing Embedded Operating Systems encounter significant roadblocks due to much less documented specifications, an inherent ineffectiveness in generating high-quality payloads. In this paper, we propose *ECG*, an Embedded Operating System fuzzer empowered by Large Language Models (LLMs) to sufficiently mitigate the aforementioned issues. *ECG* approaches fuzzing Embedded Operating System by automatically generating input specifications based on readily available source code and documentation, instrumenting and intercepting execution behavior for directional guidance information, and generating inputs with payloads according to the pre-generated input specifications and directional hints provided from previous runs. These methods are empowered by using an interactive refinement method to extract the most from LLMs while using established parsing checkers to validate the outputs. Our evaluation results demonstrate that *ECG* uncovered 32 new vulnerabilities across three popular open-source Embedded OS (RT-Linux, RaspiOS, and OpenWrt) and detected 10 bugs in a commercial Embedded OS running on an actual device. Moreover, compared to *Syzkaller*, *Moonshine*, *KernelGPT*, *Rtkaller*, and *DRLF*, *ECG* has achieved additional kernel code coverage improvements of *23.20%*, *19.46%*, *10.96%*, *15.47%*, and *11.05%*, respectively, with an overall average improvement of *16.02%*. These results underscore *ECG*'s enhanced capability in uncovering vulnerabilities, thus contributing to the overall robustness and security of the Embedded Operating System.**

***Index Terms*—Vulnerability Detection, Embedded Operating System, Fuzz Testing**

I. Introduction

With the increasing development of industrial automation, a growing number of industrial devices are empowered by Embedded Operating Systems (Embedded OSs), ranging from smart manufacturing to aeronautics and astronautics. Unlike general-purpose operating systems that are designed for daily usage, Embedded Operating System are customized to meet very specific industrial requirements, like real-time performance, low power consumption, and high reliability. Also, Embedded OSs tend to deploy on mission-critical scenarios; any security vulnerabilities could make Embedded OSs fail to deliver its services and even lead to severe consequences, such as data leakage or system crashes [1], [2]. Therefore, it is urgent to detect and fix vulnerabilities in Embedded OSs, especially for their customized modules.

Fuzz Testing, a.k.a. *fuzzing* [3]–[7], is a widely used technique for detecting software vulnerabilities, it generates a large number of test cases and feeds them into the System-Under-Test (SUT) to trigger potential bugs. It has been widely deployed to ensure the security of various software systems, including operating systems, web browsers, and network protocol implementations, with many vulnerabilities discovered over the past few years.

For fuzzing operating system kernels, the state-of-the-art is *Syzkaller* [8], which has been widely used to detect vulnerabilities in many OS kernels, including Linux, BSDs, and macOS, etc. It leverages system call specifications (Syzlang [9]) as input grammar to generate corresponding system call sequences and arguments. It is also assisted by a coverage-guided feedback mechanism, allowing *Syzkaller* to effectively select inputs with higher potential, and in turn generate high-quality test cases to uncover any potential vulnerabilities within the kernel. In particular, the system call specifications, which are manually-written by kernel experts, contain information such as the prerequisite resources, a set of system call sequences, and their corresponding arguments. The quality of these manually-written specifications has a major impact on the fuzzing performance, as it determines the path of execution the code takes within the kernel.

Motivation: To produce satisfactory fuzzing results, we need to produce high-quality input payloads that allow the flow of execution to reach wide and deep within the Embedded OS kernel's state space. Current methods are met with the following two limitations. (1) First, writing system call specifications with the quality and quantity of *Syzkaller*, while at the same time supporting many Embedded OSs, currently requires significant manual efforts and years worth of domain knowledge to accomplish. This process is especially difficult to perform adequately for Embedded OSs due to their customized and unique components, which often are not extensively documented, thus posing even more difficulties to overcome. (2) Second, generating payloads for Embedded OSs during testing is difficult for established methods, as current specifications only define correct syntax but lack its corresponding semantics, which leads to the fuzzer generating ineffective input payloads. As many Embedded OSs have individual semantics, which are not available on the scale of popular OSs (e.g., Linux), constructing these manually on a large scale is impossible to achieve.

Manuscript received March 31, 2024; revised June 16, 2024; accepted July 14, 2024. This article was presented at the International Conference on Embedded Software (EMSOFT) 2024 and appeared as part of the ESWEEK TCAD special issue.

This research is sponsored in part by the National Key R&D Program of China (2023YFB4503704), NSFC Program (No.62202500), Hunan Provincial Natural Science Foundation (No. 2023JJ40772), Open Fund of Anhui Province Key Laboratory of Cyberspace Security Situation Awareness and Evaluation (No.CSSAE-2023-010).

Q. Zhang and W. Chang are with the College of Computer Science and Electronic Engineering, Hunan University, Changsha 410082, China (e-mail: zhangqiang9413@126.com).

H. Shi is with the School of Electronic Information, Central South University, Changsha 410083, China, and Anhui Province Key Laboratory of Cyberspace Security Situation Awareness and Evaluation (e-mail: hey.shi@foxmail.com).

Y. Shen, J. Liu, Y. Xu and Y. Jiang are with the KLISS, BNRist, School of Software, Tsinghua University, Beijing 100084, China (e-mail: shenyh20@mails.tsinghua.edu.cn).

Wanli Chang is the corresponding author.

Large Language Models (LLM), known for their excellence in comprehending natural language text, are widely used to assist in various testing responsibilities. By leveraging LLMs in fuzzing Embedded OSs, we can potentially overcome these hurdles by using LLMs to generate high-quality test payloads by automatically creating system call specifications based on code and documentation in place of human writers, and interpreting both the documentation and the postmortem analysis of each test run for semantic information, allowing it to generate test payloads that contain more operation semantics, thereby improving the fuzzer's capabilities of reaching more code and runtime states. However, despite LLM's rich potential in improving testing effectiveness, a direct application is infeasible, the reasons for which are explained below.

Challenges: First, LLMs are predisposed towards texts upon which it has been previously trained, making direct specification generation inefficient. Our preliminary tests show that LLMs cannot generate working specifications reliably, which is mainly due to 1) low training data due to scarce availability, 2) low variety in the data, as it only describes the system call interfaces of a given OS, and 3) low quality of generated type and constraint information for dependent data structures due to limited context size. In contrast, LLMs are generally better at generating code in C, as proven by numerous previous attempts in the wild [10]–[12]. Also, compared to general-purpose OSs, Embedded OSs are specialized with specific requirements, thus having certain modifications, which is knowledge LLMs are not pre-trained upon. The lack of sufficient documentation further increases the difficulty of generating valid specifications.

Second, current kernel fuzzing methods are ill-equipped to generate high-quality payloads effective in testing Embedded OS components. They are hindered by their payload generation methods that do not account for semantic information. By utilizing LLMs, we are able to interpret more information both statically from its source code and documentation, as well as processing the post-mortem execution results of test runs. Practically, however, these guidance hints generally show up in the form of natural language text, such as documentation text or tracing information, which are unintelligible for established methods but are better interpreted through LLMs.

Solution: To address these challenges, we propose *ECG*, an Embedded OS fuzzer that leverages LLMs to improve Embedded OS fuzzing efficiency. First, *ECG* parses the source code of the target Embedded OS to identify interface candidates. The candidates are then statically analyzed to obtain type and constraint information of their positional arguments. This information, along with its corresponding documentation, is passed to a pre-testing LLM to generate actual code that utilizes the interface, along with some hints that the LLM produces regarding the arguments' constraints and directional information. Then, the code is refined with the LLM until it produces a passing code snippet, which is then transformed into a system call specification. The specification is then combined with the hints as attributes and compiled as input generation routines for runtime use.

During fuzzing, after the execution of each input, the execution trace is passed to the post-processing LLM, which is prompted to output whether the input payload's execution reaches a better directional vector than previous attempts at the target module. If so, the LLM's output and the input itself are saved for further mutation. When a previous input is selected for further mutation, the original input payload, the hints from static analysis, and the output from the post-processing LLM are then passed to the input generation LLM. The resulting input is then executed to obtain its execution trace, and *ECG* processes the output through the process again.

Evaluation: To understand our approach's effectiveness, we evaluate *ECG* on three widely used Embedded OSs: RT-Linux, RaspiOS, and OpenWrt. Our results show that *ECG* found 32 previously unknown bugs on 3 open-sourced Embedded OSs and 10 bugs on a commercial Embedded OS deployed on a Raspberry PI. Also, comparing with the state-of-the-art fuzzer, *Syzkaller*, *Moonshine*, *KernelGPT*, *Rtkaller*, and *DRLF*, *ECG* achieves an average of *23.20%*, *19.46%*, *10.96%*, *15.47%*, and *11.05%* improvement in code branch coverage, respectively, yielding an overall average increase of *16.02%*. This demonstrates the effectiveness of *ECG* in enhancing the fuzzing performance of Embedded OS.

Contribution: We make the following contributions:

- We implemented *ECG*, an Embedded OS fuzzer that leverages LLM to enhance the quality of generated input, thereby boosting the overall fuzzing performance.
- We propose to leverage extracted Embedded OS code and document to provide fuzzer with detailed mutation guidance. This allows the fuzzer to perceive the fuzzing direction, assisting it in exploring in-depth kernel code.
- We demonstrate the effectiveness of *ECG* through extensive evaluation on three popular open source Embedded OSs and a commercial Embedded OS. We also open source *ECG* [1] and make it available for practice.

II. Background

A. Embedded Operating System and Kernel Fuzzing

Embedded Operating Systems (Embedded OSs) are specialized operating systems designed to manage and operate hardware within various appliances and embedded devices, ranging from consumer electronics to manufacturing robots and automation controllers. Within this domain, Embedded Linux has emerged as a prominent choice due to its versatility, open-source nature, and extensive support for a variety of hardware platforms. Embedded Linux has been deployed in many applications, including automotive systems, home appliances, and industrial automation devices. Embedded Linux uses a customizable kernel, allowing developers to strip down or add components based on the specific requirements of their project, thus optimizing both system size and performance. There are many well-known embedded Linux distributions, such as RT-Linux [13], RaspiOS [14], and OpenWrt [15], each with its own unique features and functionalities. For instance, RT-Linux has a real-time extension, such as PREEMPT_RT, which equips embedded Linux to meet the deterministic response times required by real-time applications in industrial

[1] *ECG* is available at: https://github.com/zzqq0212/ECG

settings. The OpenWrt distribution, on the other hand, is designed for wireless routers and other network devices.

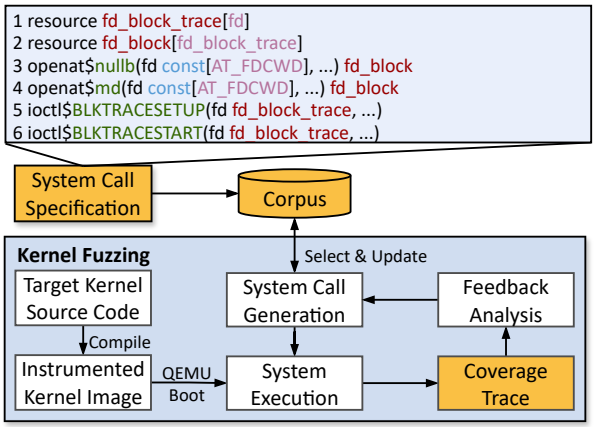


Fig. 1: Overall workflow of kernel fuzzing.

Fuzz testing, a.k.a. *fuzzing*, is an automated software testing technique that constructs a large number of malformed payloads as test input, feeds them into the System-Under-Test (SUT), and monitors unexpected behaviors, such as crash or hang, to detect potential vulnerabilities. Kernel fuzzing has been widely used to detect vulnerabilities in the kernel. *Syzkaller* [8] is a prominent example of a state-of-the-art kernel fuzzer. Taking *Syzkaller* for example, it uses system call specifications for input generation, as shown in the upper part of Figure 1. The system call specification defines prerequisite resources (Lines 1-2), defines a set of system call declarations (Lines 3-6), and *specialized* system calls that represent those requiring certain arguments (Lines 5-6).

B. LLM Assisted Testing

Large Language Models (LLMs) [16]–[18] are artificial neural networks that have excellent abilities in understanding and generating human-like text. Aside from simple linguistic interactions, they also have demonstrated their remarkable capability in solving a wide range of software engineering tasks, such as generating various programming language codes, comprehension of code documentation, and code analysis. Recently, some academic works have also leveraged the power of LLMs to improve the effectiveness of software testing, whose efforts have shown promising results [19]–[24].

One such approach is LLM-assisted fuzzing, which combines traditional fuzz testing approaches with the LLMs' abilities in interpreting data. This includes tasks like input grammar generation or complex payload synthesis, which are too complex for established methods. By generating test cases or improving the quality of fuzz inputs based on the understanding of code semantics and structure provided by LLMs, this hybrid approach can lead to more effective and efficient detection of vulnerabilities. LLMs can also assist in analyzing fuzz testing results, categorizing bugs, and suggesting fixes, making the entire process more intelligent and less labor-intensive. This integration of LLM capabilities with fuzz testing methodologies represents a promising frontier in the quest to enhance software security and reliability.

III. Motivation

While current fuzzing approaches have achieved results in finding bugs, one of the factors significantly affecting the bug-finding capabilities of fuzzers is the quality of the input generated, which is mainly dependent on the input grammar for generation-based methods and the capabilities of payload generation techniques in generating semantic-rich input payloads. In *Syzkaller*'s case, it uses system call specifications as input grammar, where most of the specifications are written by kernel experts, which is time-consuming and requires years' worth of domain knowledge. While its generation method uses a rule-based routine that utilizes specifications to ensure syntactically correct input generation, the lack of semantics is partially mitigated through the use of *specialized* descriptions, which decompose system calls into their functional subsets, thus restricting the input generation space and improving semantic richness.

However, when fuzzing the numerous Embedded OSs, these methods become infeasible due to the requirement for intensive manual efforts and certain specific domain knowledge to analyze the kernel's source code and extract relevant specifications. The situation is further complicated by Embedded OS's customized functionalities, which are maintained by different vendors, where its coding style and the way it uses system calls differ from one another, making it harder to construct system call specifications using current manual techniques.

LLMs, despite their very recent inception, have demonstrated significant capabilities in generating high-quality text and code. Furthermore, they can substantially reduce manual efforts in analyzing and handling complex information such as source code and documentation. Intuitively, this allows us to provide LLMs with Embedded OS documentation and its corresponding source code target modules' code areas.

Based on this, we can further extract corresponding system call specifications for the fuzzer, thereby greatly reducing the manual efforts involved in constructing specifications and improving Embedded OS fuzzing performance. Additionally, we can also prompt the LLM to output semantic hints regarding the usage of these system calls, including type and constraint information regarding its positional arguments, as well as required processing system calls or subsequent calls. Similarly, we can also use LLMs to interpret execution feedback, such as stack trace and code coverage, to guide input payload generation further. Processing this information for usage semantic information, usually in natural language form, requires us to prompt the LLM once more and obtain the resulting payload from its output.

To effectively leverage LLMs in generating high-quality test inputs, we may still face the following two challenges: First, LLMs are much less effective at generating an operating system's system call specifications than commonly written routines in programming languages (e.g., C, Python, etc.). This is mainly due to 1) LLMs' training efforts mainly focusing on the plethora of general programming languages available, such as common algorithms and programming paradigms in C, Python, etc., 2) a lack of variety in the data learned, as it only describes a handful of system calls from few OSs, and 3)

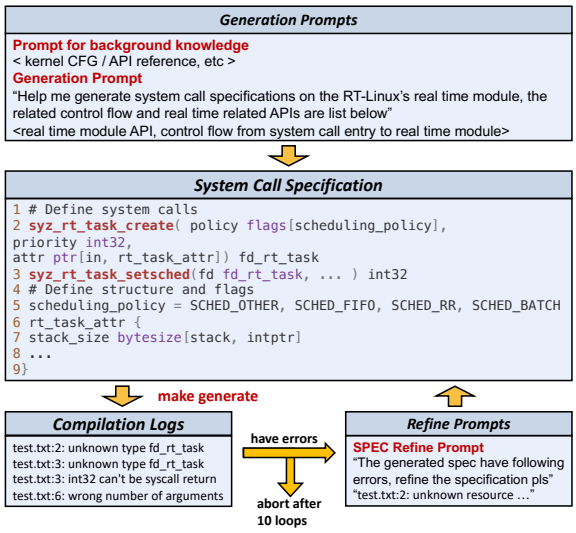


Fig. 2: The erroneous system call specifications generated by LLM in real-time module.

limited context size that restricts its effectiveness in generating all dependent data structure definitions and the system call specifications. Additionally, Embedded OSs contain specialized components that are not constituent components of the upstream kernels, and therefore LLMs do not have the relevant information trained. These modules are also generally not as well-documented as the upstream kernel documentation, thus presenting more difficulties for LLMs. We demonstrate such an issue with the example given in Figure 2, where we feed ChatGPT's *GPT-4* model with a prompt including Embedded OS's background knowledge and related generation prompt and pass its resulting system call specifications to *Syzkaller*'s specification compiler, which reports many errors within. This is not easily rectifiable, as we continuously refined the output by feeding the compiler's error logs back to the LLM, which then returned more erroneous outputs.

Second, current kernel fuzzer cannot efficiently generate payloads with correct semantics that effectively test the state space of Embedded OSs' components. Current state-of-the-art methods, such as those used in *Syzkaller*, still rely on rule-based pseudo-random generation and mutation techniques, which cannot leverage semantic hints generated during system call specification generation or after interpreting a single execution round's behavior, such as code coverage. While Linux has specifically benefited from the implementation of *specialized* system call specifications that represent partial semantics, types, and value assignments, this approach is not feasible for other Embedded OS due to limited documentation and the extensive effort required for a single Embedded OS. Additionally, these hints, generated by LLMs, are typically presented in natural language text, which traditional fuzzing methods can hardly interpret or comprehend.

Therefore, to generate a high-quality payload for Embedded OS, we can take a small detour from the original goal. As shown in Figure 3, instead of directly generating system call specifications, we first use the LLM to generate C code tailored for the target Embedded OS. This step leverages

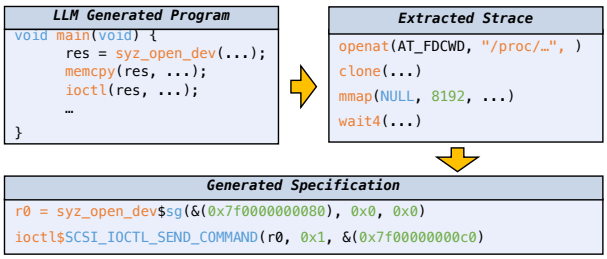


Fig. 3: Input generation example.

LLM's capability in generating commonly used programming language. After obtaining the C code, we employ `strace` to capture execution trace information, which provides a detailed view of the system calls made during execution. We then use `trace2syz` to convert these traces into structured system call specifications. During the fuzzing process, we enhance raw coverage data to extract more meaningful semantic information. This enables the LLM to interpret the system's behavior better and improve the generation of input payloads that are both relevant and effective. This method not only streamlines the generation process but also ensures the payloads are accurately aligned with the target system's requirements.

IV. ECG Design

In this section, we present the design of *ECG*, an LLM-powered Embedded Operating System fuzzer to automatically generate input specifications for a given Embedded OS and produce higher-quality input payloads to effectively test the embedded components of the given Embedded OS.

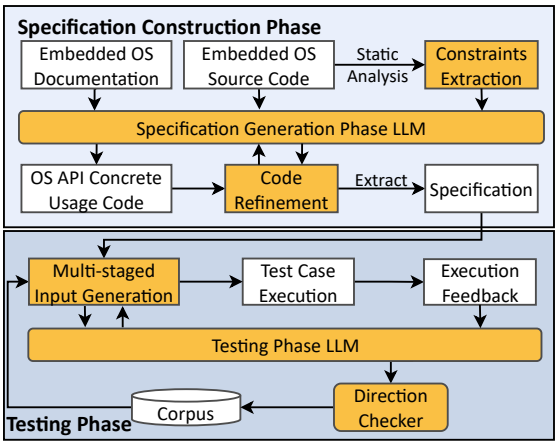


Fig. 4: Overview of *ECG*. *ECG* utilizes two LLMs in its workflow, which consists of a *Specification Generation* phase and the *Testing* phase.

We show the overall architecture of *ECG* in Figure 4. As shown, *ECG* consists of two main phases: *Specification Construction Phase* and *Testing Phase*. During its *Specification Construction* phase, *ECG* first extracts system call definitions within the source code and conducts static analysis to extract argument type and constraint information. This, along with its source code and corresponding documentation, is then

passed to the *Specification Generation Phase* LLM, which is prompted to output a working example of such a system call, along with hints regarding its argument ranges and constraints. The resulting code is *refined* by continuously checking the code using a compiler and feeding any error messages back into the LLM for further generation. Based on the valid test programs, *ECG* then extracts the corresponding system call specifications and uses them as input payload generation grammar during fuzzing. When the fuzzing campaign is underway, i.e., the *Testing Phase*, *ECG* leverages a *Testing Phase* LLM that generates test payloads from constructing new system call sequences according to their specifications and hints, or by mutating existing input payloads using hints extracted from static code and from interpreting the results of previous runs. The input payload is then executed on the target Embedded OS, which is monitored by *ECG*. After each execution, the execution trace and coverage data are then extracted and passed back to the LLM, which is then prompted to interpret the quality of the input payload by assessing its execution novelty and vector to its target module. This is then used to refine the hints associated with each test case in the corpus as well as its corresponding specification.

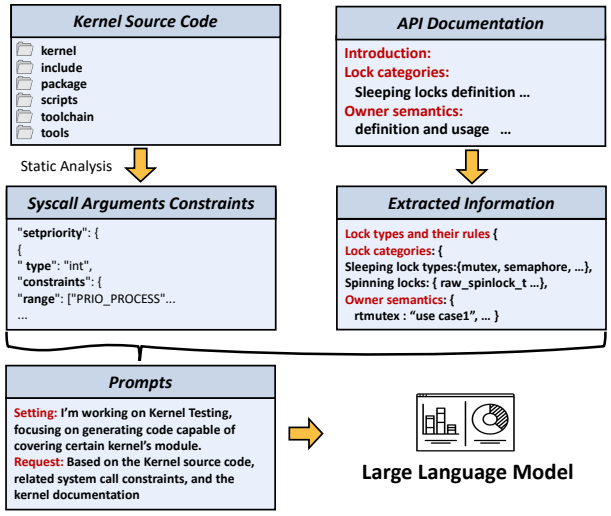


Fig. 5: The overall workflow of obtaining constraints of system calls for a target Embedded OS.

A. Specification Construction

To construct the Embedded OS's system call specifications for the target Embedded OS, we mainly use the Embedded OS's *documentation* and *source code*, and with the help of an LLM, generate valid C code that utilizes the system calls and reach the relevant embedded modules, and convert the C program into specifications for the system call. Since an off-the-shelf LLM struggles to generate syntactically and semantically accurate code for an Embedded OS on which it has not been trained. We adopt a specification generation pipeline that mainly consists of the following three steps: 1) source-level static analysis, 2) actual usage code generation and refinement by LLM, and 3) specification extraction from actual usage code.

Giving LLMs a deep understanding of the target Embedded OS's internals is a challenging task due to the large code scale and the complex interaction within the code logic. LLMs inherently struggle to comprehend long texts, particularly in extensive codebases such as those of an Embedded OS. Furthermore, it is not intensively trained with the target Embedded OS, therefore is further at a disadvantage.

Therefore, we first perform static analysis to extract more constraint information to reduce the difficulties LLMs have and increase the success rate of obtaining a valid specification. Starting with the documentation and source code of the target Embedded OS, we search the code base for system call declarations and cross-reference the documentation for validation. These are then statically analyzed, where the arguments of each system call are inspected and assigned attributes depending on their explicit type definitions. The attributes, such as possible flag values or range demarcation values for integer values, along with the system call definitions themselves, as well as the corresponding documentation page, are constructed into a prompt for the LLM, where it is asked to compose a piece of actual working code that leverages this system call interface to reach the modules referenced in the Embedded OS documentation.

Algorithm 1: Constraint Extraction Algorithm

```
   Input: s = Source Code of Embedded OS
   Input: d = Embedded OS Documentation
   Input: p = System Call Matching Pattern
1  Function AnalyzeArg(arg):
2      attrs ← ∅
3      switch typeof(arg) do
4          case Integer do
5              attrs[flag_val] ← flag_analyze(arg)
                attrs[range_val] ← rang_analyze(arg)
6          case Pointer do
7              attrs[is_in], attrs[is_out] ←
                dir_analyze(arg)
8          case Structure do
9              for member ∈ arg do
10                 attrs[member] ←
                    analyze_arg(member)
11     return attrs
12 Function extract_constraints():
13     syscalls ← [x for x in regex.match(s, p)]
14     for syscall ∈ syscalls do
15         args ← syscall.args
16         for arg ∈ args do
17             arg.attrs ← AnalyzeArg(arg)
18             return args
```

*1) **Constraint Extraction**:* This constraint extraction process is depicted in Figure 5. As shown in the diagram, we take the source code and documentation of an Embedded Operating System, parse the code for system call declarations, extract the system calls and their corresponding descriptions, perform static analysis to obtain the constraints, and pass it to the *Specfication Phase Generation LLM*. The algorithm is also shown in Algorithm 1. The process begins at line 12. As shown in line 13, we parse and find system call definitions by lexical analysis using regular expressions.

The regular expressions are crafted manually, as these system call declarations have uniform patterns. In line 16, we extract constraints for each declared argument. The way the constraints are extracted is dependent on the type of the positional argument. In the case of structures (line 8), we recurse into each of its member variables. The constraints are then attributed to the target system call and prepared for processing by the LLM.

*2) **Program Generation and Refinement**:* We feed the *Specification Generation Phase* LLM with the extracted constraints, along with its original source code snippet, as well as its corresponding documentation page, and prompt it to output a working C code that utilizes the system call, with appropriate arguments that are valid and allow the flow of execution to reach the kernel sections mentioned in the documentation, if possible. The initial program given by the LLM can be erroneous, or its execution cannot proceed into the target modules or sections. We use *program refinement* to resolve these issues, mainly by compiling and executing the generated program, analyzing if any errors were raised during the process, and feeding this information back to the LLM to prompt for another generation attempt.

Algorithm 2: Program Generation and Refinement

```
Input: S : System Call Source Code
Input: D : System Call Documentation
Input: C : System Call Constraints
Output: P : System Call Specifications
  e ← ∅ // Set of errors collected
  i ← K // Number of tries left
  a ← nil // Generated program
  do
      i ← i − 1
      p ← LLM.PromptGen(S, D, C, e)
      a, e ← Compile(p)
      if a ≠ nil then
          e ← e ∪ ForkAndExec(a)
  while e ≠ ∅ ∧ i ≥ 0
  for s ∈ a do
      p ← Code2Spec(s)
      P ← P ∪ p
  return P
```

The overall process of the specification construction phase is concisely written in Algorithm 2. The algorithm takes the source code and documentation of the target system call, as well as the constraints obtained in the previous step, as input to the LLM. It then prompts the LLM to output a working program in the programming language specified by Embedded OS and assign arguments that are able to execute into, if it is provided, the target component within the Embedded OS kernel. Any errors raised during code generation, compilation, and execution will be fed back into the LLM for further generation assistance. The algorithm sets an upper limit $\mathbf{K}$ to the number of iterations to prevent infinite loops.

*3) **Specification Extraction**:* After obtaining a working program, we extract a system call specification, along with its dependent data structures and hints regarding its usage and positional argument value assignment, and compile relevant generation routines for the fuzzer's input payload generation mechanisms to use during runtime. The input generation payload hints are embedded into the specifications as attributes and thus can be picked up by the generation mechanisms. If errors are raised in this process, we repeat the previous step to obtain another working program and reattempt this process.

B. Runtime Input Payload Generation

For the fuzzing process to keep generating high-quality seeds and further enhancing fuzzing performance, we leverage a *Testing Phase* LLM in assisting seed generation. Existing state-of-the-art kernel fuzzers like *Syzkaller* do not have the ability to interpret highly informative attributes containing hints, as they purely depend on code coverage information and rule-based randomized generation methods to guide the fuzzer to generate test cases. However, while LLMs are competent at processing natural languages, they cannot understand unprocessed binary information well. To assist LLMs in understanding the execution feedback and suggesting the best mutation direction, we can convert the binary coverage data into more semantic-rich information. Therefore, in the generation phase, the mutation direction insights guide LLM in the iterative creation of arguments that adhere to system call specifications.

*1) **Direction Checker**:* Currently, coverage feedback information used by kernel fuzzers is expressed as a bitmap, where the index is the processed address of the basic block, and the value is the number of times the basic block is executed. Without information like kernel code disassembly and corresponding coverage calculation techniques, LLMs cannot easily understand such information to assist in input payload generation. To use LLMs to interpret the execution results, we take coverage formatted in hexadecimal, along with the Embedded OS's control flow graph and current execution trace logs, into a structured and pseudo-natural-language written format, and prompt the LLM to output a possible distance vector, which can directly show the best mutation direction to mutate the system call's arguments. The details of this process are discussed as follows.

When each execution round finishes, we collect the coverage information and decompose the binary kcov trace. Since the kernel contains the address for each kernel symbol, we can get the current execution trace and the executed times for each block based on the address in kcov. Then, by combining the trace with the target module's address, we can calculate the distance between the current execution trace and the target module. The distance to a target module is determined by counting the intermediate basic blocks on the shortest path from a starting block, for example, on the trace $BB_1 \rightarrow BB_2 \rightarrow BB_3$, the distance from BB_1 to BB_3 is 2. we define an array representing the distance of every basic block within the target module to the closest point in the execution trace. The module distance is then computed as the average of these distances, quantifying how far the current execution is from fully covering the target module.

This computation can be formalized as follows: Let $D = d_1, d_2, \ldots, d_n$ be the set of distances where d_i is the distance from the i^{th} basic block in the target module to the nearest

point in the execution trace. The module distance, MD, is calculated as the mean of all values in D. This module distance MD is a heuristic for guiding the overall fuzzing process. Lower values of MD indicate closer proximity to the target module, pointing to areas of potential interest for a more detailed exploration. Based on MD, we introduce a *direction vector* for each argument in the system calls. This vector suggests how the argument might mutate to potentially decrease MD, drawing the execution trace to the target module. The direction vector V for an argument encompasses components that include:

- **Add/Subtract** $(+/-)$: Suggests whether to increase or decrease numeric arguments.
- **Multiply/Divide** $(*/\div)$: Indicates whether scaling the argument up or down is preferable.
- **Flag Toggle**: For arguments acting as flags, suggests alternating between different flag values.

The selection of a mutation operation is influenced by both the argument type and the current module distance, aiming to explore mutations more likely to reduce MD effectively. This vector is then treated as the program's attributes and will guide the multi-staged generation phase to generate more meaningful test payloads.

*2) **Multi-staged Generation**:* We use LLM to either mutate existing input payloads by directional hints accumulated from previous runs or choose proper system calls and generate arguments within the system call specifications. However, LLMs have a limited context size and are therefore incapable of handling long texts that exceed this limit, while each specification contains a different number of arguments, and each argument has specific possible values. Consequently, generating all arguments simultaneously may cause the fuzzer to produce relatively low-quality test cases.

To address such problems, we use a multi-staged generation process, where we first let LLM choose best-fit system calls for this execution round, then when mutating the arguments, we leverage the direction vector to guide LLM to generate the arguments within the system call specifications. However, instead of generating the system call as a whole, we separate the generation task into small tasks, i.e., generate each argument in isolation and combine them to complete the system call's instantiation. Specifically, after we provide the initial set of corpus seeds and the kernel's control flow graph to the LLM, the LLM selects the most promising system call based on its potential to navigate closer to target areas within the kernel. Then, for each argument of this syscall, the algorithm calculates a direction vector to guide its mutation, ensuring each modified argument aligns with strategic exploration objectives. Furthermore, utilizing the LLM, we generate arguments that are not only contextually relevant but also optimized for penetrating deeper into the kernel's logic. Last, these arguments are incrementally assembled into a refined system call, which is then added to the next program to be tested against the kernel.

V. Implementation

We implemented *ECG* using a mixture of code written in Golang, Python, and C++. We first use LLVM [25] to compile the Embedded OS code and use *passes* to scan the Embedded OS's code, lowered into LLVM's Intermediate Representation (IR). This process allows us to identify the target module's entry function and backpropagate possible control flows from the system call entry to the module's entry function. We collect the functions and corresponding arguments constraint for each control flow and then organize them into JSON files to facilitate the specification construction phase.

To generate code for specification extraction, we choose *Mixtral-8x7b* [26] as our *Specification Construction Phase* payload generation LLM due to its code generation abilities and comprehension skills. We prompt *Mixtral-8x7b* with necessary Embedded OS source code and documentation, allowing it to establish a basic understanding of the generation target. Then, we use a Python script to provide *Mixtral-8x7b* with text file for code generation. Once we collect the valid generated programs, we extract the execution trace for each program using *strace* and convert it into system call specifications using conversion utilities provided by Moonshine [27].

To generate input payloads, we use *Mistral-7b* [28] as the *Testing Phase* interpretation and payload generation LLM due to its adequate generation qualities and fast token generation speed. In detail, we intercept the raw *KCOV* coverage, combine it with the disassembled kernel image, and translate the coverage information into a text file to inform LLM which functions have been covered. Then, we calculate the distance from the current direction to the target module. Furthermore, combined with the corresponding input, processed coverage, and distance information, we use LLM to provide mutation hints, including mutation direction and system call priority. Leveraging such information, during the payload mutation phase, we use LLM to iteratively generate arguments within the system call specifications and use the generated arguments as the test payloads to test the target Embedded OS further.

VI. Evaluation

To understand our approach's effectiveness, we propose the following research questions to help us understand *ECG*'s performance and effectiveness.

- **RQ1:** Is *ECG* able to uncover new bugs in target Embedded OS?
- **RQ2:** Is *ECG* capable of covering more code sections than other tools?
- **RQ3:** What is the effectiveness of the LLM-assisted generation strategy?

A. Evaluation Setup

The experiments were conducted on a server with a 128-core AMD EPYC CPU, 32 GiB of memory, and 2 Tesla V100S-PCIE-32GB GPUs running Ubuntu as the host kernel. We selected the RT-Linux, RaspiOS, and OpenWrt as our test target, as these three Embedded OS are mostly widely used Embedded OS. Also, for the target versions, we selected kernel versions 6.7 and 6.8 for RT-Linux and RaspiOS, respectively, along with versions 5.15 and 6.1 for OpenWrt, as these versions of kernel are the latest stable releases at the time of our experiments. Each testing target uses the same compilation configuration; the KCOV and KASAN options are enabled

TABLE I: *ECG* has identified 32 previously undiscoverd bugs within the latest Linux kernel.

#	Modules	Versions	Locations	Bug Types
1	fs/buffer	RT-Linux 6.7	mark_buffer_dirty	logic error
2	drivers/pci	RT-Linux 6.7	vga_put	logic error
3	kernel/sched	RT-Linux 6.7	select_task_rq_fair	deadlock
4	mm/filemap	RT-Linux 6.7	filemap_fault / page_add_file_rmap	data race
5	drivers/net/	RT-Linux 6.7	e1000_update_stats	memory corruption
6	fs/inode	RT-Linux 6.7	inode_update_timestamps	data race
7	kernel/kprobes	RT-Linux 6.7	arch_adjust_kprobe_addr	logic error
8	fs/dcache	RT-Linux 6.7	d_splice_alias	data race
9	fs/ext4	RT-Linux 6.7	__ext4_new_inode /_find_next_zero_bit	data race
10	drivers/e1000	RT-Linux 6.7	e1000_clean	data race
11	lib/find_bit	RT-Linux 6.7	_find_first_bit	data race
12	kernel/sched	RT-Linux 6.8	__wake_up_common	null-ptr defer
13	lib/kasprintf	RT-Linux 6.8	kvasprintf	logic error
14	kernel/events	RT-Linux 6.8	perf_cgroup_switch	logic error
15	fs/inode	RT-Linux 6.8	generic_update_time / inode_needs_update_time	data race
16	fs/ext4	RT-Linux 6.8	generic_write_end / mpage_submit_folio	data race
17	fs/kernfs	RT-Linux 6.8	kernfs_dop_revalidate	memory corruption
18	fs/ext4	RT-Linux 6.8	ext4_split_extent_at	memory corruption
19	arch/x86/lib	RT-Linux 6.8	memcpy_orig	out-of-bounds
20	kernel/events	RT-Linux 6.8	free_event	logic error
21	drivers/scsi	RT-Linux 6.8	__bitmap_weight /scsi_device_unbusy	data race
22	kernel/rcu	RT-Linux 6.8	__call_rcu_common /mas_walk	data race
23	mm/swap	RT-Linux 6.8	__folio_end_writeback / lru_add_fn	data race
24	arch/x86/lib	RT-Linux 6.8	memmove	memory corruption
25	arch/x86/events/intel	RT-Linux 6.8	intel_pmu_lbr_counters_reorder	logic error
26	arch/x86/kernel	RT-Linux 6.8	deref_stack_reg	logic error
27	arch/x86/kernel	RT-Linux 6.8	__orc_find	memory leak
28	net/9p	RT-Linux 6.8	p9pdu_readf	memory leak
29	arch/x86/lib	OpenWrt 5.15	memset_erms	logic error
30	kernel/smp	OpenWrt 5.15	smp_call_function_single	logic error
31	arch/arm64/kvm	RaspberryPi OS 6.7	kvm_init_stage2_mmu	memory leak
32	arch/arm64/kvm	RaspberryPi OS 6.7	kvm_age_gfn	logic error

for coverage collection and memory corruption detection. For thread-related issues, such as data race vulnerabilities, we use the KCSAN configuration.

We used *Syzkaller*, *KernelGPT*, *Moonshine*, *Rtkaller*, and *DRLF* for comparison against *ECG*. Furthermore, we propose *ECG*-, which is *ECG* without LLM guided generation for component-wised comparison and *ECG*-directed, which is *ECG* with directed fuzzing enabled for comparison with *DRLF*. The experiment maintained consistent parameters, including QEMU resources and fuzzer instances. Specifically, to control the computational resources, we started all experiments simultaneously and distributed the resources evenly, including 2 cores and 2 GB of main memory for each virtual machine. To ensure the accurate statistic of *ECG*'s effectiveness and to mitigate potential biases, each experiment is repeated 10 times over 24-hour periods, with the resultant data being averaged to determine the final outcomes. The empirical evaluation referred to the prudent evaluation practices for fuzzing [29].

B. Bug Detection Capabilities

To answer **RQ1** and demonstrate the bug detection ability of *ECG*, we use *ECG* to test target Embedded OSs for a week. In the conducted experiments, the *ECG* identified 32 previously unknown bugs. Despite rigorous testing with extensive computing resources by established kernel fuzzers such as *Syzkaller*, these bugs remained undiscovered. It is noteworthy that most of the bugs were located within the foundational logic of the kernel, specifically affecting critical modules, including file systems, network protocols, and process schedule modules. Comprehensive details regarding these bugs, including the affected kernel module, version, precise location, and bug types, are listed in Table I.

The ability of *ECG* to disclose previously unknown bugs relies on its specification construction and LLM-assisted fuzzing strategies. The specification construction strategy allows LLM to generate C programs that access the targeted kernel module by extracting the anchoring Embedded OS's internals constraint so that it is further processed into high-quality corpus seeds for Embedded OS testing. Additionally, the direction checker and multi-staged generation mechanism of LLM assisted fuzzing optimize the mutation direction guidance of generating the arguments within the system call specification.

Bug Impact. In particular, we investigate the impact of bugs identified by the *ECG*, which manifest in various adverse outcomes such as data corruption, system freezes, and system crashes. The analysis focuses on the consequences of the identified vulnerabilities. Fifteen newly discovered bugs are primarily related to logic errors and memory corruption, which could lead to data corruption or loss. These bugs are categorized as such: bugs 1, 2, 5, 7, 13, 14, 17, 18, 20, 24, 25, 26, 29, 30, and 32. Furthermore, an additional 12 bugs, primarily associated with deadlocks and data race issues, have been discovered. These bugs, which include bugs 3, 4, 6, 8, 9, 10, 11, 15, 16, 21, 22, and 23, may cause the system to hang or become unresponsive. Moreover, we have identified five new bugs, mainly concerning buffer overflows and memory leaks, which could lead to system crashes and functional failures. including bugs 12, 19, 27, 28, and 31. If exploited carefully,

these bugs will reduce the system's reliability and may cause it to break down completely.

TABLE II: Average bug counts of *ECG* and other fuzzers over 10 rounds of experiments.

Metrics	OpenWrt		RT-Linux		RaspiOS		Total
	v5.15	v6.1	v6.7	v6.8	v6.7	v6.8	
ECG	7.1	7.8	8.9	8.2	6.5	7.4	45.9
ECG-	6.3	7.2	7.8	7.3	5.8	6.4	40.8
KernelGPT	5.5	6.3	6.7	6.1	5.2	5.7	35.5
Moonshine	4.5	4.9	4.4	5.1	4.2	4.7	27.8
Rtkaller	4.7	5.3	5.2	5.5	4.6	5.1	30.4
Syzkaller	4.1	4.6	4.3	4.8	3.9	3.7	25.4
ECG-directed	4.3	4.6	3.9	4.2	4.1	3.5	24.6
DRLF	3.5	4.1	3.2	3.7	3.6	3.1	21.2

Bug Detection Comparison. We compare the bugs triggered during testing. As illustrated in the Table II, *ECG* identified the highest number of bugs under identical experimental conditions and duration. Specifically, *ECG* discovered the most bugs, identifying 7.1, 7.8, 8.9, 8.2, 6.5, and 7.4 bugs in OpenWrt5.15, OpenWrt6.1, RT-Linux6.7, RT-Linux6.8, RaspiOS6.7, and RaspiOS6.8, respectively. *ECG-* also discovered a total of 40.8 bugs. This is significantly due to the specification construction and runtime input payload generation in *ECG*, which effectively assists the LLM in generating input payloads capable of triggering specific kernel modules within embedded systems.

```
1  static kprobe_opcode_t * _kprobe_addr(
2     kprobe_opcode_t *addr,
3     const char *symbol_name,
4     unsigned long offset,
5     bool *on_func_entry)
6  {
7           ...
8     addr = arch_adjust_kprobe_addr((unsigned
           long)addr, offset, on_func_entry);
9     if (addr)
10       return addr;
11 }
12
13 kprobe_opcode_t *arch_adjust_kprobe_addr(
14       unsigned long addr,
15       unsigned long offset,
16       bool *on_func_entry)
17 {
18   if (is_endbr(*(u32 *)addr)) {
19     *on_func_entry = !offset || offset == 4;
20     if (*on_func_entry)
21       offset = 4;
22   } else {
23     ...
24     }
25 }
```

Fig. 6: Code snippet of memory corruption in RT-Linux probe subsystem.

Case Study. Here, we take Bug#16 as the case study to further illustrate the effectiveness of *ECG*. In detail, Figure 6 demonstrates a logic error bug found in the RT-Linux v6.7 kernel's core modules. This bug is found by *ECG* and has been confirmed by the corresponding maintainer. Concretely speaking, this bug is located in the function *arch_adjust_kprobe_addr()*(Line 13), which is called by the kernel's real-time module, when it uses *pref_events()* to periodicity monitor the kernel's behavior. The bug is triggered due to the premature dereferencing of the address pointed to by *addr* function parameter without ensuring that this address is safe and valid for access. Specifically, when the kernel invokes the call instruction *addr = arch_adjust_kprobe_addr()* (Line 8) within the *_kprobe_addr()* function, the *arch_adjust_kprobe_addr()* function then attempts to determine whether the address points to an ENDBR instruction, a part of Indirect Branch Tracking (IBT) on x86 architectures, by directly accessing the memory at that address using *is_endbr(*(u32 *)addr)* (Line 18). This operation assumes that the address is valid and has been mapped into the process's address space. However, if the address is invalid, unallocated, or does not have the appropriate permissions for access, this dereferencing leads to a segmentation fault or general protection fault, causing the kernel crash.

To trigger this bug, *ECG* extracted constraints from the targeted Embedded OS kernel's real-time documentation to generate payloads targeting the real-time modules for the LLM. Furthermore, it transformed the coverage feedback information obtained during the fuzzing process into LLM-readable structured data to guide the optimization of mutation directions. As a result, *ECG* successfully triggered this bug. Although there has been considerable testing work on Embedded OS and their real-time module, *ECG* distinguishes itself from previous testing efforts by generating precise seed corpus through the integration of function constraint dependencies and LLM code generation. Additionally, with the LLM aid of mutation direction guidance feedback, it can produce seed cases that explore given target system modules.

Real Device Testing. We extended our testing framework to real-world Embedded OS. Specifically focusing on the ZHIXIN OS, on the Raspberry Pi platform. ZHIXIN OS is a Embedded OS widely used in electronics systems to monitor and control power supply. We conducted tests using two Raspberry Pi devices, each running ZHIXIN OS, with multiple fuzzing instances deployed. The setup consists of a laptop and the Raspberry Pis. The laptop, using a LLM-generated corpus, manages the fuzzing process. It connects to the Raspberry Pis via GPIO communication, transferring fuzzing data, including test cases, coverage, and bug reports. The Raspberry Pis runs ZHIXIN OS, injecting test cases at startup. They collect coverage and kernel dmesg in real-time, sending it back to the laptop for analysis. Parallel execution distributes test cases across the Raspberry Pis, efficiently covering a broad spectrum of the OS's functionality. During the fuzzing campaign on ZHIXIN OS, we discovered 10 bugs,

TABLE III: ZHIXIN OS bugs found by *ECG*.

Modules	Operations	Bug Types
arch/x86/kernel	orc_ip	null-ptr deref
kernel/workqueue	process_one_work	data race
arch/x86/kernel	profile_pc	use-after-free
lib/iov_iter	_copy_from_iter	logic error
fs/buffer	generic_write_end / mpage_process_page_bufs	data race
kernel/fork	alloc_pid / copy_process	data race
fs/ext4	ext4_bmap	deadlock
fs/ext4	ext4_mb_good_group / mb_mark_used	data race
fs/stat	generic_fillattr / kernfs_refresh_inode	data race
fs/proc	task_dump_owner	data race

TABLE IV: Average branch coverage and p-value statistics of *ECG* and other fuzzers over 10 rounds of experiments.

Subject	OpenWrt		RT-Linux		RaspiOS		Overall
	v5.15	v6.1	v6.7	v6.8	v6.7	v6.8	
ECG	**160210.6**	**173660.5**	**206004.3**	**215922.5**	**176111.4**	**168805.2**	**183,452.4**
ECG-	151153.5(+5.99%/0.014)	166260.5(+4.55%/0.022)	196004.3(+5.10%/0.035)	203922.5(+5.88%/0.025)	165550.9(+6.38%/0.011)	151640.6(+11.32%/0.026)	**172422.1(+6.4%)**
Syzkaller	130876.4(+22.41%/0.001)	144863.5(+19.88%/0.003)	171117.3(+20.39%/0.009)	172977.7(+24.83%/0.003)	139509.9(+26.24%/0.003)	134090.6(+25.89%/0.005)	**148,905.8(+23.20%)**
Moonshine	135201.3(+18.50%/0.007)	148661.9(+16.82%/0.004)	174380.2(+18.14%/0.013)	178459.3(+20.99%/0.007)	145825.3(+20.77%/0.008)	138891.7(+21.54%/0.008)	**153,569.9(+19.46%)**
KernelGPT	143582.5(+11.58%/0.011)	160346.4(+8.30%/0.012)	185393.4(+11.12%/0.021)	193854.7(+11.38%/0.015)	158850.5(+10.87%/0.01)	149928.1(+12.59%/0.013)	**165,325.9(+10.96%)**
Rtkaller	139634.2(+14.74%/0.001)	153689.7(+12.99%/0.003)	179253.7(+14.92%/0.016)	184209.4(+17.22%/0.006)	151840.7(+15.98%/0.005)	144622.3(+16.72%/0.006)	**158873.6(+15.47%)**
ECG-directed	**47228.6**	**49845.2**	**46180.7**	**44865.5**	**47228.6**	**49856.5**	**47534.1**
DRLF	43283.3(+9.12%/0.023)	44646.5(+11.64%/0.017)	41099.7(+12.36%/0.012)	40445.2(+10.93%/0.041)	43585.6(+8.36%/0.011)	43754.4(+13.95%/0.024)	**42802.4(+11.05%)**

highlighting the effectiveness of *ECG* in finding hidden bugs. Detailed bug statistics are in Table III. These bugs are mainly in core modules like the file system, network, and process management. Bug types include null-pointer dereference, use-after-free, deadlock, memory leak, data race, and logic error. If exploited, they could cause ZHIXIN OS to crash or lose data. We responsibly disclosed our findings to the developers and maintainers of ZHIXIN OS, with 2 bugs confirmed.

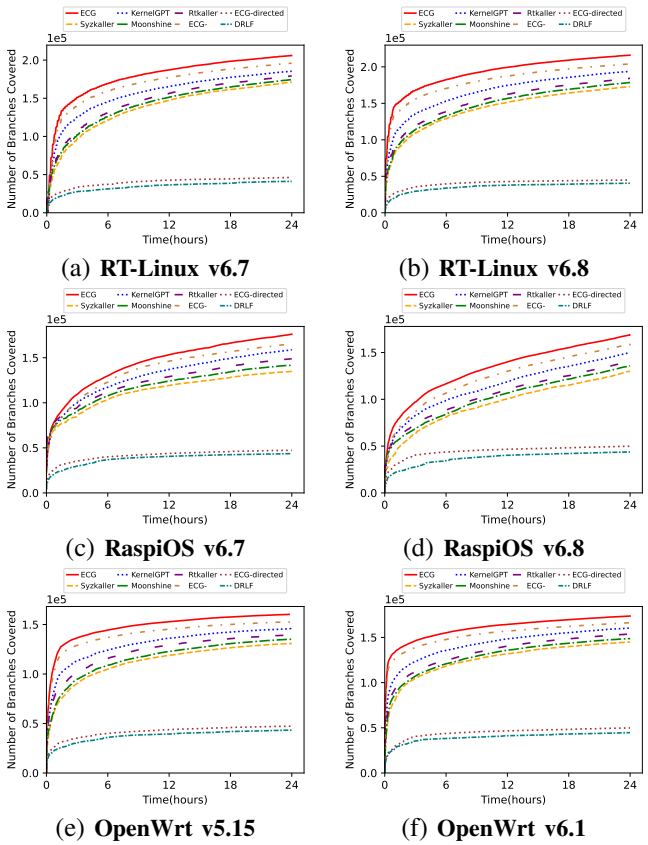
C. Coverage Improvement of ECG

To answer **RQ2** and demonstrate the effectiveness of *ECG* in exploring a broader range of execution paths and deeper kernel states, we present the code coverage statistics in comparison with existing state-of-the-art kernel fuzzers, including *Syzkaller*, *Moonshine*, *KernelGPT*, *Rtkaller*, and *DRLF*. 6 different embedded Linux kernel versions are chosen for the experiment, including RT-Linux and RaspiOS Kernel versions 6.7 and 6.8, along with OpenWrt Kernel versions 5.15 and 6.1. The Qemu simulation environments utilized by all fuzzer tools maintain the same parameter configurations, such as CPU core number, memory storage size, and so on. Finally, to minimize statistical deviation, each experiment is repeated 10 times.

Table IV provides a detailed breakdown of the branch coverage statistics for *Syzkaller*, *Moonshine*, *KernelGPT*, *Rtkaller*, *DRLF*, *ECG-*, and *ECG* over a 24 hours period. Compared to these tools, *ECG* achieves average coverage improvements of *23.20%*, *19.46%*, *10.96%*, *15.47%*, *11.05%*, and *6.4%*, respectively. These results show that *ECG* not only achieves higher coverage on all kernel versions but also significantly enhances the exploration depth of the embedded operating system kernel code, thereby indicating its superior effectiveness in identifying potential bugs.

We further plot the coverage growth curve based on the coverage statistic. As illustrated in Figure 7, the initial phase of the evaluation demonstrates a continuous increase in coverage for all evaluated tools. Notably, *ECG*, *ECG-*, and *KernelGPT* exhibit a growth rate surpassing that of *Syzkaller*, *Moonshine*, and *Rtkaller* in the early stages. Furthermore, approximately after the first three hours, coverage expansion of the *ECG* and *ECG-* consistently outpaces that of *KernelGPT*, maintaining this lead until the conclusion of the experimental period.

A significant benefit of *ECG* is its utilization of LLM to generate source code programs that interact with specific modules of the embedded operating system, thereby generating a higher quality initial corpus than other fuzzers. Additionally, *ECG* mutates high-quality corpus seeds under the guidance of structured feedback from the fuzzing loop, facilitating rapid penetration into the deeper layers of the embedded operating



(a) **RT-Linux v6.7** (b) **RT-Linux v6.8**

(c) **RaspiOS v6.7** (d) **RaspiOS v6.8**

(e) **OpenWrt v5.15** (f) **OpenWrt v6.1**

Fig. 7: Average coverage growth curve comparison of *ECG* and other fuzzers.

system kernel logic. In comparison, *Syzkaller* initiates its process with an arbitrary sequence of system calls, often overlooking complex kernel code paths. Similarly, *Rtkaller* and *DRLF* face the same challenge, lacking high-quality seed sequences that effectively guide the fuzzing process toward critical code modules. *Moonshine* uses sequences derived from actual programs, but these generally fail to trigger the deep kernel's code. Moreover, *KernelGPT* produces syscall specifications that are unsuitable for embedded operating systems and struggles to adapt in such scenarios.

Comparsion with Tardis. Since *Tardis* does not support Embedded Linux testing and supports those systems such as uc/os and FreeRTOS, we adapted *ECG* to support FreeRTOS testing for comparison. In detail, we feed FreeRTOS's API documentation to the LLM, allowing *ECG* to generate test cases for it, and we adapted *Syzkaller* to support the target

RTOS fuzzing. [2] In summary, *ECG* and *ECG-* achieve an average of 4205.3 and 4097.2 branch coverage, with a previously unknown bug found in the esp_memory_utils module. In comparison, *Tardis* achieves an average of 3888.0 branch coverage, which is 8.16% less than *ECG*.

D. Effectiveness of LLM-Assisted Generation

To answer **RQ3**, we evaluate the effectiveness of the LLM-assisted generation strategies in terms of the generate program count, the generation efficiency, and the overhead. We chose real-time, memory management, file system, and network as the target modules, as these modules are specially tailored to the embedded environment and need to balance functionality with the limited computing resources available. For instance, the real-time module provides scheduling and timing mechanisms that allow an embedded OS to offer deterministic behavior and real-time performance; while the file system module is optimized for the specific types of storage media used in embedded devices, such as NAND flash.

TABLE V: Extracted code statistic in 3 target Embedded OSs.

Modules	RT-Linux	RASPiOS	OpenWRT	SUM
real-time	69	–	–	69
mem management	23	59	64	146
file system	43	64	23	130
network	29	52	73	154
SUM	164	175	160	499

Generation Program Statistic. The detailed generated program counts for each module are listed in Table V. In RT-Linux, generation focused on the real-time module with 69 programs, while memory management, file system, and network had 23, 43, and 29 programs, respectively. For RaspiOS, *ECG* generated 59 programs in memory management, 64 in the file system, and 52 in the network, totaling 175 programs. In OpenWrt, 64, 23, and 73 programs were generated in these modules, totaling 160 programs. Overall, 499 programs were generated across all modules and systems, highlighting that *ECG* enhances LLM's ability to produce test cases for diverse system functionalities.

Generation Efficiency. To evaluate the code generation efficiency of *ECG*, we compared *Mixtral8x7b* (used by *ECG*), *Llama3-8b*, and *GPT3.5* in terms of the averaged number of valid programs generated, repair programs, and token consumption for each LLM within 10 minutes, over 10 repetitions, with the same prompts.

TABLE VI: Average number comparison of *Mixtral8x7b*, *Llama3-8b*, and *GPT3.5* on generated/repaired programs, and the tokens consumption per program in 10 minutes over 10 rounds of experiments.

Subject	Valid/Gen	Repaired/Invaild	Token Consumption
Mixtral8x7b	10.2/18.6	4.7/8.4	781.9
Llama3-8b	6.7/14.3	3.9/7.6	824.1
GPT3.5	9.1/17.8	4.1/8.7	857.4

The results are shown in Table VI. *Mixtral8x7b* generated 18.6 programs, with 10.2 directly executable and 8.4 non-executable. After LLM-guided repairs, 4.7 programs were successfully repaired. In comparison, *Llama3-8b* and *GPT3.5* generated 14.3 and 17.8 programs, with 6.7 and 9.1 being executable. Additionally, *Mixtral8x7b* consumed fewer tokens on average, approximately 781.9 tokens. This efficiency is due to its enhanced syntactic and semantic understanding of programming and efficient tokenization, allowing for more directly executable programs with fewer tokens, reducing computational resource usage, and improving the debugging and repair process.

Runtime Overhead. We further evaluated the memory and runtime overhead of the LLM used in the *ECG* for the experiments. The results indicate that *ECG* uses an average of 28.32 GB of memory per graphics card when executing a single prompt in the code generation phase, while each graphics card uses an average of about 13.61 GB when executing a single prompt in the testing guidance phase. We also measured the average time required to generate an input payload, which was 32.25 seconds.

VII. Related Work

In the domain of operating system kernel fuzzing [30], [31], *Syzkaller* has been widely used to detect vulnerabilities in the Linux kernel. *Syzkaller* focuses mainly on general purpose OSs, where *ECG* is aimed towards Embedded OSs. Many works try to enhance the fuzzing performance of *Syzkaller*. For example, *Healer* [32], a fuzzer inspired by *Syzkaller*, uses a relation learning algorithm to deduce the relation between different system calls and use such information to guide the test case generation. In comparison, *ECG* aims to generate input payloads by automatically constructing input specifications and identifying semantic information from static documentation and code, as well as dynamic sources from postmortem analysis of execution rounds.

For Embedded OS fuzzing, many works attempted to improve the fuzzing performance of Embedded OS [30], [33]. For example, *Tardis* [30] proposes the use of shared-memory mechanisms to collect execution coverage, thus guiding input generation thereby boosting real-time operating system fuzzing performance. *DRLF* [34] uses a directed feedback mechanism to guide the test case generation and drive the fuzzing toward real-time related code sections. These works mainly focus on leveraging runtime feedback to guide the fuzzing process while failing to provide the fuzzer with high-quality input payloads. *ECG*, in contrast, interprets semantic information with the help of LLMs to further assist in the input payload generation process. Additionally, *ECG* also proposes an automated specification generation technique that provides the foundational input grammar.

Empirically, initial seeds play a pivotal role in improving kernel fuzzing performance. As such, many works proposed techniques to obtain an initial seed corpus and increase its quality. For instance, *Moonshine* [27] collects the execution trace in the real world and uses a distillation algorithm to refine the quality of generated input. *KernelGPT* [24] leverages LLM to generate the initial corpus for fuzzer. In contrast to *ECG*, *KernelGPT* does not provide any insight into generating specifications for Embedded OS, nor does it improve the input semantics generated from existing techniques.

[2]https://github.com/zzqq0212/ECG/tree/main/ECG/ecg-freertos

VIII. Discussion

Generation Performance. In the current landscape of employing LLM for test case generation, one significant hurdle that stands out is the generation speed of these models. The intricate nature of LLM, designed to comprehend and generate complex code structures, inherently demands considerable computational resources and time. To mitigate the impact of this limitation on the fuzzing workflow, we have adopted an asynchronous approach, allowing test case generation to proceed without stalling the overall testing process. In the future, we can adapt lightweight LLMs tailored for tasks like code generation. Additionally, *ECG*'s reliance on well-documented files limits its applicability in environments with poor or outdated documentation, which is common in legacy embedded systems. To improve applicability, we can enhance *ECG* with man-in-the-loop techniques or refined strategies to extract interface information, which can then be used by LLMs to generate test cases, reducing reliance on documentation.

LLM in Bug Detection. Currently, LLM for Embedded OS fuzzing predominantly focuses on payload generation. This narrow application range underscores the current limitations of LLM, particularly in areas crucial to fuzzing, such as bug detection and feedback analysis. Although the existing methods can generate complex payloads, they can't analyze fuzzing feedback, such as identifying subtle bugs. In the future, LLMs could be trained for improving bug detection.

IX. Conclusion

In this paper, we present *ECG*, an LLM-assisted Embedded OS fuzzer. By leveraging LLM to generate high-quality payloads, *ECG* can enhance Embedded OS fuzzing performance. *ECG* found 32 previously unknown bugs on 3 open-sourced Embedded OSs, achieves *16.02%* coverage improvement on average, compared with the state-of-the-art fuzzers. We also deployed *ECG* on a commercial Embedded OS, ZHIXIN OS, running on actual hardware, where it found 10 bugs.

References

[1] A. Greenberg, "A decade-old bug is putting millions of critical devices at risk," 7 2019. [Online]. Available: https://www.wired.com/story/vxworks-vulnerabilities-urgent11/

[2] C. Cimpanu, "Zephyr rtos fixes bluetooth bugs that may lead to code execution," 12 2020. [Online]. Available: https://www.bleepingcomputer.com/news/security/zephyr-rtos-fixes-bluetooth-bugs-that-may-lead-to-code-execution/

[3] S. Mallissery and Y.-S. Wu, "Demystify the fuzzing methods: A comprehensive survey," *ACM Comput. Surv.*, vol. 56, no. 3, oct 2023. [Online]. Available: https://doi.org/10.1145/3623375

[4] "Aflgo: Directed greybox fuzzing." [Online]. Available: https:https://github.com/aflgo/aflgo

[5] P. Chen and H. Chen, "Angora: Efficient Fuzzing by Principled Search," in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018.

[6] M. Wang, J. Liang, C. Zhou, Z. Wu, X. Xu, and Y. Jiang, "Odin: on-demand instrumentation with on-the-fly recompilation," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 1010–1024. [Online]. Available: https://doi.org/10.1145/3519939.3523428

[7] M. Wang, J. Liang, C. Zhou, Y. Jiang, R. Wang, C. Sun, and J. Sun, "RIFF: Reduced instruction footprint for Coverage-Guided fuzzing," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, Jul. 2021, pp. 147–159. [Online]. Available: https://www.usenix.org/conference/atc21/presentation/wang-mingzhe

[8] D. Vyukov and A. Konovalov, "Syzkaller: an unsupervised coverage-guided kernel fuzzer," 2015, https://github.com/google/syzkaller.

[9] ——, "Syzlang: System call description language," 2015, https://github.com/google/syzkaller/blob/master/docs/syscall_descriptions_syntax.md.

[10] L. Plein, W. C. Ouédraogo, J. Klein, and T. F. Bissyandé, "Automatic generation of test cases based on bug reports: a feasibility study with large language models," 2023.

[11] Z. Zhang, C. Chen, B. Liu, C. Liao, Z. Gong, H. Yu, J. Li, and R. Wang, "Unifying the perspectives of nlp and software engineering: A survey on language models for code," 2024.

[12] B. Chen, F. Zhang, A. Nguyen, D. Zan, Z. Lin, J.-G. Lou, and W. Chen, "Codet: Code generation with generated tests," 2022. [Online]. Available: https://arxiv.org/abs/2207.10397

[13] "Real-time linux kernel." 2010, https://wiki.linuxfoundation.org/realtime/start.

[14] "Raspberrypi os." 2012, https://www.raspberrypi.com/documentation/computers/os.html.

[15] "Openwrt." 2004, https://openwrt.org/.

[16] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, and e. Harri Edwards, "Evaluating large language models trained on code," 2021.

[17] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt, C. C. Ferrer, A. Grattafiori, W. Xiong, A. Défossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, and G. Synnaeve, "Code llama: Open foundation models for code," 2024.

[18] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong, Y. Du, C. Yang, Y. Chen, Z. Chen, J. Jiang, R. Ren, Y. Li, X. Tang, Z. Liu, P. Liu, J.-Y. Nie, and J.-R. Wen, "A survey of large language models," 2023.

[19] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software testing with large language models: Survey, landscape, and vision," 2024.

[20] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," 2024.

[21] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang, "Large language models for software engineering: Survey and open problems," 2023.

[22] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. 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*, ser. ISSTA 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 423–435. [Online]. Available: https://doi.org/10.1145/3597926.3598067

[23] Z. Yuan, Y. Lou, M. Liu, S. Ding, K. Wang, Y. Chen, and X. Peng, "No more manual tests? evaluating and improving chatgpt for unit test generation," 2023.

[24] C. Yang, Z. Zhao, and L. Zhang, "Kernelgpt: Enhanced kernel fuzzing via large language models," *arXiv preprint arXiv:2401.00563*, 2023.

[25] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, ser. CGO '04. USA: IEEE Computer Society, 2004, p. 75.

[26] MistralAI, "Mixtral-8x7B-v0.1," 2023. [Online]. Available: https://huggingface.co/mistralai/Mixtral-8x7B-v0.1

[27] S. Pailoor, A. Aday, and S. Jana, "MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 729–743. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/pailoor

[28] MistralAI, "Mistral-7B-v0.1," 2023. [Online]. Available: https://huggingface.co/mistralai/Mistral-7B-v0.1

[29] M. Schloegel, N. Bars, N. Schiller, L. Bernhard, T. Scharnowski, A. Crump, A. Ale-Ebrahim, N. Bissantz, M. Muench, and T. Holz, "Sok: Prudent evaluation practices for fuzzing," in *2024 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2024, pp. 140–140. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP54263.2024.00137

[30] Y. Shen, Y. Xu, H. Sun, J. Liu, Z. Xu, A. Cui, H. Shi, and Y. Jiang, "Tardis: Coverage-guided embedded operating system fuzzing," *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 41, no. 11, pp. 4563–4574, 2022. [Online]. Available: https://doi.org/10.1109/TCAD.2022.3198910

[31] Y. Xu, H. Sun, J. Liu, Y. Shen, and Y. Jiang, "Saturn: Host-gadget synergistic usb driver fuzzing," in *2024 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2024, pp. 50–50. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP54263.2024.00051

[32] H. Sun, Y. Shen, C. Wang, J. Liu, Y. Jiang, T. Chen, and A. Cui, *HEALER: Relation Learning Guided Kernel Fuzzing*. New York, NY, USA: Association for Computing Machinery, 2021, p. 344–358. [Online]. Available: https://doi.org/10.1145/3477132.3483547

[33] Y. Shen, H. Sun, Y. Jiang, H. Shi, Y. Yang, and W. Chang, "Rtkaller: State-Aware Task Generation for RTOS Fuzzing," *ACM Trans. Embed. Comput. Syst.*, vol. 20, no. 5s, sep 2021. [Online]. Available: https://doi.org/10.1145/3477014

[34] Y. Shen, S. Chen, J. Liu, Y. Xu, Q. Zhang, R. Wang, H. Shi, and Y. Jiang, "Brief industry paper: Directed kernel fuzz testing on real-time linux," in *2023 IEEE Real-Time Systems Symposium (RTSS)*. Los Alamitos, CA, USA: IEEE Computer Society, dec 2023, pp. 495–499. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/RTSS59052.2023.00059