Tardis: Coverage-Guided Embedded Operating System Fuzzing

Yuheng Shen, Yiru Xu, Hao Sun, Jianzhong Liu, Zichen Xu, Aiguo Cui, Heyuan Shi✉ and Yu Jiang✉

***Abstract*—Embedded Operating Systems are extensively deployed in many mission-critical industrial scenarios. Any defects within these systems may result in unacceptable losses. Therefore, it is imperative to develop tools to detect bugs within Embedded Operating Systems, thus minimizing potential impacts on industrial infrastructures. Coverage guided fuzzing is a vulnerability detection technique that has found numerous real-world vulnerabilities within both application programs as well as kernels. However, state-of-the-art kernel fuzzers, e.g., *Syzkaller*, mainly target general purpose operating systems, such as Linux, macOS, and Windows, whereas Embedded Operating System support is mostly lacking. In this paper, we propose *Tardis*, the first Embedded Operating System fuzzer capable of testing a wide selection of Embedded Operating Systems while leveraging coverage feedback. *Tardis* conducts OS-agnostic code coverage collection and analysis, allowing developers and testers to test a wide range of Embedded Operating Systems without significant manual efforts. We implemented and evaluated *Tardis* on several well-known Embedded Operating Systems, such as UC/OS and FreeRTOS. *Tardis* can successfully perform fuzz testing on these kernels without significant manual effort for adaptation. By leveraging coverage feedback, *Tardis* can cover 51.32% more branches than black-box fuzzing on average on the respective Embedded Operating Systems over 24 hours. *Tardis* also found 17 previously unknown bugs among the target Embedded Operating Systems.**

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

I. Introduction

EMBEDDED Operating Systems (Embedded OSs) are operating systems that perform certain tasks in specific environments. Compared to general purpose operating systems, they specialize in carrying out specific tasks and provide more stability. Currently, these operating systems have been extensively deployed in various industrial settings, including many mission-critical scenarios, such as the aerospace, mining and automotive manufacturing industries. The security of these systems is of paramount importance, since many of its bugs may result in catastrophic consequences. Fuzz testing (fuzzing) is an automated software bug detection technique that has recently been applied to test operating system kernels. For example, *Syzkaller* [28], one of the most prominent kernel fuzzing tools (fuzzers), has identified thousands of vulnerabilities up until this date.

Essentially, fuzzers test programs by feeding different inputs to the program-under-test and monitoring the execution of the program for any abnormal and unexpected behavior. However, non-trivial programs, especially operating systems, have an input space that is too large to be explored thoroughly. In order to efficiently explore a program's input space and possible points for bugs, fuzzers generally leverage coverage feedback as an indicator of the program's state after accepting the generated input. Collecting coverage feedback is generally a two-part procedure. First, testers instrument the target program with routines to collect coverage statistics at runtime. Second, the fuzzer analyzes the trace statistics after executing the program and identifies whether the input can trigger any new and "interesting" behavior within the target program. If so, the fuzzer saves the relevant input as a starting point for further exploration into the program's state. This technique greatly increases fuzzing efficiency, as it drastically reduces the input space required to explore.

State-of-the-art kernel fuzzers also employ coverage feedback, commonly by following the design paradigms of user-land coverage guided fuzzers. For instance, *Syzkaller* leverages coverage feedback in Linux to guide further input generation and mutation through the following procedures. First, it builds the Linux kernel with *KCOV* [25] enabled. Then, *KCOV* reports coverage statistics from the target kernel at runtime, which can be retrieved by *Syzkaller*. *Syzkaller* identifies and preserves test cases that trigger new kernel execution paths via analyzing the gathered coverage statistics. Compared to fuzzing without coverage feedback, coverage-guided fuzzing can explore the target program's state space more efficiently and be more effective in finding vulnerabilities.

However, the current coverage collection and analysis procedures are usually tightly coupled to specific operating systems or architectures due to their design and implementation requirements. This imposes significant obstacles when attempting to adapt feedback mechanisms for Embedded Operating Systems. Specifically, the main difficulties are as follows. First, Embedded Operating Systems run on a variety of architectures and platforms, thus preventing fuzzers from utilizing hardware support for coverage collec-

Manuscript received April 07, 2022; revised June 11, 2022; accepted July 05, 2022. This article was presented at the International Conference on Embedded Software (EMSOFT) 2022 and appeared as part of the ESWEEK-TCAD special issue.

This research is sponsored in part by the NSFC Program (No. 62022046, 92167101, U1911401, 62021002, 62192730), National Key Research and Development Project (No. 2019YFB1706203, No2021QY0604) and MIIT Project(Design of intelligent networked vehicle based on SOA central control).

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

H. Shi is with the Big Data Institution, Central South University, Changsha 410083, China (e-mail: hey.shi@foxmail.com).

Aiguo. Cui is with the Godel Lab, Huawei Technologies Co., Ltd, Shanghai 200001, China (e-mail: ag.cui@huawei.com).

Z. Xu is with the School of Mathematics and Computer Science, Nanchang University, Nanchang 330031, China (e-mail: xuz@ncu.edu.cn).

Yu Jiang and Heyuan Shi are the corresponding authors.

tion, such as Intel PT, as used in kAFL [21]. Several works propose to utilize binary instrumentation into Embedded OS fuzzing for coverage guidance, e.g., *Gustave* [6]. However, this kind of approach is highly coupled with the target architecture, thus requiring immense engineering efforts to enable coverage guided fuzzing for different architectures. Thus, coverage feedback mechanisms for Embedded Operating Systems should be implemented using source code based instrumentation. Second, commonly used Embedded Operating Systems are numerous and their interfaces are diverse, resulting in the difficulty of utilizing a unified coverage collection interface based on existing designs. In contrast, most general purpose kernel fuzzers leverage *KCOV* to instrument the target kernel and collect coverage information. However, adapting *KCOV* to Embedded Operating System kernels and utilizing them during fuzzing relies heavily on kernel-specific operations, such as I/O and memory mapping. To the best of our knowledge, however, many popular Embedded Operating Systems do not support the complete suite of interfaces required. Meanwhile, different Embedded OSs normally provide different interfaces for user land, thus a unified, OS-agnostic design is needed for coverage collection. Finally, the coverage collection and analysis mechanisms should be efficient so that they would not significantly impact the execution throughput of the fuzzer.

Therefore, to facilitate coverage-guided fuzzing across a wide selection of popular embedded operating systems, we shall address the following challenges. First, the coverage feedback mechanisms should be software-based and avoid relying on hardware-specific mechanisms to guarantee portability. Second, these mechanisms should be OS-agnostic, i.e. it is able to operate without relying on any OS-specific features and can adapt to different OSes without additional costs. Third, the mechanisms' implementations should be highly efficient and avoid introducing any significant overheads.

To address the above challenges, we propose *Tardis*, a coverage-guided Embedded Operating System kernel fuzzer. *Tardis* conducts coverage guided fuzzing using the following approaches. First, *Tardis* utilizes a *bitmap*-based storage for coverage collection, which is OS-agnostic so that the procedure can be utilized for a wide range of Embedded OSs. In order to guarantee portability among different architectures, *Tardis* chooses to instrument the coverage collection procedure into Embedded OSs during compilation. For each executed code branch, *Tardis* records it into a shared coverage buffer with several simple *bitwise* and *shift* instructions, which guaranteeing the efficiency of collection. The coverage buffer is shared between the host fuzzer and the guest though exposing partial memory space of *QEMU* instance, so that the potential overhead of coverage data transferring to the host is eliminated. Finally, *Tardis* analyzes the collected coverage statistic on the host also with simple instructions in a CPU cache friendly manner, which promises the efficiency analysis on coverage data.

We evaluated *Tardis* on several widely used Embedded OSs. The results show that *Tardis* achieves a 51.32% improvement in branch coverage on average compared with black-box fuzzing. *Tardis* also found 17 previously unknown bugs in target Embedded OSs.

In summary, we make the following contributions:

- We propose *Tardis*, a coverage guided Embedded OS fuzzer. It can instrument on an Embedded OS and run at the bare-metal level while conducting highly efficient fuzzing by generating high-quality test cases based on runtime coverage information.
- We design a coverage collection mechanism that is OS-agnostic and cross-architecture, allowing for efficient coverage feedback on a wide variety of Embedded Operating Systems.
- We implemented *Tardis* and evaluated its effectiveness on several widely used Embedded Operating Systems. The results shows that *Tardis* found 17 new bugs. Compared with black box fuzzing, *Tardis* achieves 51.32% coverage improvement on average.

II. Background and Motivation

A. Embedded Operating Systems

Embedded Operating Systems are operating systems designed to run specialized tasks in embedded environments. They are widely used in many mission-critical situations, such as the aerospace industry and autonomous vehicles. Some widely-used Embedded OSs include FreeRTOS [2], UC/OS [13], VxWorks [16], etc. Generally speaking, Embedded OSs are designed for specific architectures, where the hardware is typically resource-constrained. In other words, an Embedded OS is optimized to improve the efficiency of managing the hardware resources and reduce response times specifically for tasks running upon it. Therefore, it is typically designed to be compact and resource-efficient. Meanwhile, it reduces many unneeded features and interfaces to improve the predictability of its runtime behavior.

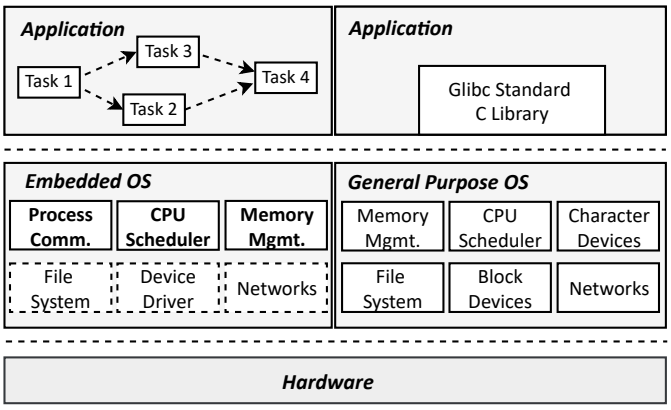


Fig. 1. Diagram of the differences between Embedded OSs and General Purpose OSs. On the hardware layer, an Embedded OS usually supports a wide range of architectures in embedded scenarios, where the hardware resources are limited, while a General Purpose OS mainly supports architectures with relatively rich hardware resources. Generally, an Embedded OS only provides core functionalities, while a General Purpose OS aims to be more comprehensive than the former.

The major differences between Embedded OSs and General Purpose OSs are shown in Figure 1. Specifically, the runtime structure of a modern operating system typically consists of three layers. The bottom layer represents the hardware on top

of which the kernel runs. It consists of components like the processor, ROM, RAM, buses, etc. Generally speaking, the architecture of a General Purpose OS that runs in desktop or server scenarios is hardware with relatively rich resources. However, an Embedded OS usually has a rather complex application scenario, e.g., industrial control and IoT devices. Therefore, the CPU that Embedded OSs run on top of can vary considerably. Additionally, the memory space provided for an Embedded OS can be significantly less, compared with a General Purpose OS. Second, an Embedded OS usually provides core functionalities only, e.g., task scheduling, memory management, with unnecessary parts cut down, while General Purpose OSs aims to be comprehensive in terms of system services. In detail, Embedded OS provides basic abstractions as an operating system environment for the user's convenience, e.g., task as a basic calculation unit and priority-based schedule. However, since Embedded OSs are mainly designed to support certain kinds of tasks with specialized purposes, many complex functionalities, e.g., network stacks and file systems, that are normally provided by General Purpose OSs, are usually removed for better predictability of the OSs' runtime behavior and reduction in its code size. These functionalities are only available in certain scenarios depending on the task requirements. Furthermore, the runtime models are different between Embedded OSs and General Purpose OSs. Specifically, the Embedded OS is designed to integrate its code with user land task-specific code into an application as a single executable image, while a General Purpose OS strictly separates the kernel space and user space as well as isolates resources between user processes. Since fuzzing requires a great deal of resources and high-level functionalities provided by the operation system, the above difference poses great difficulties in conducting the fuzz testing on Embedded OSs.

B. Coverage Guided Kernel Fuzzing.

Fuzz testing (fuzzing) is an automated vulnerability detection technique. Its basic idea is to detect a program's abnormal behavior by feeding it repeatedly with different inputs. However, randomly generated test cases are not sufficient for effective fuzzing because they can hardly accommodate the program's strict formatting requirements for inputs. To this end, coverage guided fuzzing proposes using the coverage information to identify those test cases capable of discovering previously unseen paths, then utilize them to generate higher-quality test cases, enabling efficient fuzzing. Concretely, a standard coverage guided fuzzing complies with the following procedures. First, the fuzzer instruments the target program using LLVM SanitizerCoverage or AFL-GCC. Then, within the fuzzing loop, it generates test cases that conform to the target program's input format, e.g., network packets or system calls. Furthermore, it feeds this input to the target program, detecting any abnormal behaviors with the assistance of a variety of sanitizers and assertions. Last, the fuzzer monitors its execution trace during the execution, identifies any test case that triggers new coverage, and saves it for further mutation.

The modern operating system kernel usually contains a considerable codebase; Linux, for example, has around 27.9 million lines of code on its latest release. It is inevitable for the kernel to have a large portion of vulnerabilities buried in its codebase. Therefore, many works have attempted to port fuzzing into kernel vulnerability detection [10], [21], [28], [34]. Current kernel fuzzers generally follow the design paradigms of user land coverage guided fuzzers.

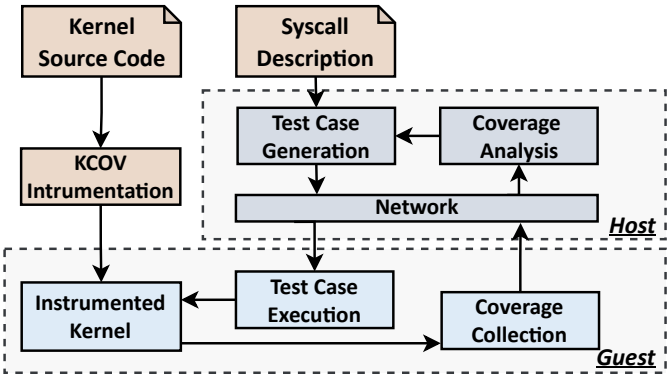


Fig. 2. Diagram of *Syzkaller* overview. It first instrument the target kernel with *KCOV*. Then, it utilizes the system call descriptions as fuzzing inputs. By parsing these system call descriptions, *Syzkaller* will generate actual system calls as test cases and send them to the target kernel. After each execution, *Syzkaller* collects the corresponding coverage information on the guest side and transfers it back on the host side for further analysis. If a test case triggers new code coverage, *Syzkaller* will mark it as interesting and save it for further test case generation.

Taking *Syzkaller* for instance, Figure 2 shows the overview of *Syzkaller*. As a coverage guided kernel fuzzer, it performs the coverage guidance by instrumenting the target kernel at compile time. Specifically, *Syzkaller* uses the system call descriptions as fuzzing inputs, which consist of the detailed information of system calls. In concrete, each system call sequence contains multiple system calls with the precise type information of their corresponding arguments and some complex resources like data structures and unions. By parsing these descriptions, *Syzkaller* can generate high-quality test cases to test the kernel. Moreover, *Syzkaller* uses the network stack like TCP/IP protocol to transfer fuzzing-related data such as test cases and the coverage information between the host and guest. After execution, *Syzkaller* will collect a test case's corresponding edge coverage, where the edge is an abstraction of the program's control flow, and send it to the host for further analysis. However, the aforementioned works are mainly designed for GPOS scenarios, and, to the best of our knowledge, none of them can conduct coverage guided Embedded OS fuzz testing.

C. Challenges in Embedded OS Fuzzing

State-of-the-art kernel fuzzers can perform adequately well on general purpose operating systems. However, they experience difficulties when adapting to Embedded OSs due to the design and implementation requirements of relevant mechanisms. Specifically, there are three major challenges that need to be addressed in order to perform coverage-feedback fuzzing on Embedded OSs.

First, Embedded OSs target a wide variety of architectures and system boards, whereas general purpose operating systems

generally target only a handful of instruction set architectures. For instance, UC/OS supports over 50 architectures, each with different hardware capabilities and resources, while Linux, one of the most widely used multi-architecture general purpose operating systems, only runs on around 10 ISAs, with varying degrees of support. This largely prevents embedded operating system kernel fuzzers from leveraging hardware features to assist in fuzzing-relevant operations such as coverage collection. Furthermore, a diverse set of supported architectures significantly increases the difficulty involved in conducting binary instrumentation, as each architecture would require individual care and effort to work correctly.

Second, Embedded OSs generally do not conform to a unified standard that provides consistent interfaces to interact with the kernel. general purpose operating systems such as Linux, BSD, and macOS are POSIX-compliant, allowing fuzzers to easily leverage well-established libraries and facilities. Fuzzers for general purpose operating systems also generally leverage KCOV [25] to collect coverage statistics. However, as demonstrated in Figure 3, porting KCOV to these embedded operating systems requires leveraging relevant kernel-specific facilities and operations, such as file systems, memory mapping, network stacks, etc. This requires significantly more human effort, as well as potential changes to the kernel itself, which can also introduce false positives, i.e., previously non-existent bugs and vulnerabilities.

Third, coverage collection and feedback mechanisms should be highly efficient, thus reducing the impact on the fuzzer's throughput. For the target Embedded OS, kernel fuzzers need to instrument code for coverage collection at a specific location into the target binary, e.g., at the beginning of each basic block. The injected code calculates each executed branch and records it into the coverage storage, thus is executed frequently and can present an immense amount of extra instructions. Since Embedded OSs normally run in a hardware environment with limited resources, we need to guarantee the efficiency of instrumented coverage collection code. For the kernel fuzzer itself, the coverage analysis is a hot path during the whole fuzzing campaign that is conducted after each test case execution. Consequently, the efficiency of the feedback mechanism can highly impact the fuzzing throughput.

Therefore, we propose the following design principles to adapt coverage feedback mechanisms for embedded operating systems: 1) the mechanisms should be purely software-based, thus avoiding any reliance on hardware features; 2) the mechanisms should be OS-agnostic, i.e., perform coverage collection and analysis regardless of the target operating system; 3) the mechanisms should exhibit a low runtime overhead, thus guaranteeing the fuzzer's overall efficiency.

III. TARDIS Design

Figure 4 shows the overall workflow of *Tardis*. First, the Embedded OS source code is cross-compiled by Clang, and the OS-agnostic coverage collection code is instrumented into the target OS for better portability, which utilizes the instrumentation capability provided by Clang compiler. *Tardis* then utilizes a modified version of QEMU to boot the target OS,

Coverage Initialization

```
struct file_operations kcov_fops = {
  .open            = kcov_open,
  .unlocked_ioctl  = kcov_ioctl,
  .mmap            = kcov_mmap,
  .release         = kcov_close,
};
```

Coverage Buffer Open

```
func kcov_open(struct inode *inode, struct file *filep) {
  struct kcov *kcov;
  kcov = kzalloc(sizeof(*kcov), GFP_KERNEL);
    ...
  spin_lock_init(&kcov->lock);
    ...
}
```

Fig. 3. Code snippet of *KCOV*. The *KCOV* mainly contains four operations, namely coverage open, I/O, sharing, and free, respectively. We further take a code snippet of kcov_open(), which is used to initialize the coverage collection. It first allocate a data buffer for coverage storage and uses threading operations to operate the coverage data.

during which partial memory space of QEMU is exposed to the fuzzer so that the coverage buffer can be directly accessed. The coverage collection assigns each basic block of the target OS with a unique ID for code branch identification during the OS booting phase. For each test case, the executed code branch is collected on the fly based on the assigned ID and is stored in the shared coverage buffer. *Tardis* then compares the collected coverage with the global coverage bitmap in the host, the size of which is compact enough to fit into the CPU cache, to detect and preserve interesting test cases with simple instructions like *bitwise* for efficiency. In this way, *Tardis* can facilitate coverage-guided fuzzing across a wide selection of popular embedded operating systems.

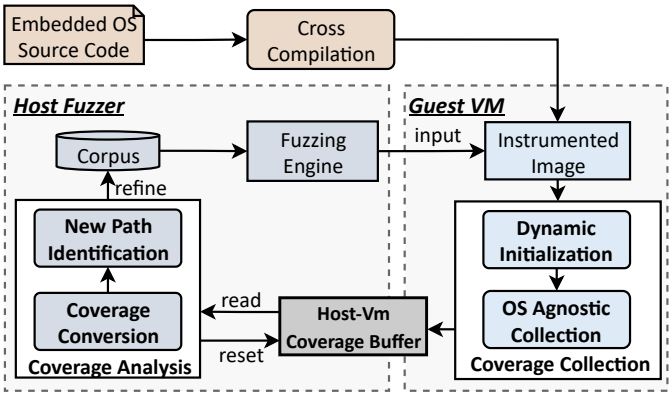


Fig. 4. Overall workflow of *Tardis*. The Embedded OS source code is cross-compiled by Clang, it outputs the target OS image, with the OS-agnostic coverage collection functions instrumented. During the OS booting phase, *Tardis* uses a modified QEMU to set up a Host-VM coverage buffer for coverage sharing, and it dynamically initializes coverage collection by giving each basic block an ID. During fuzzing, the coverage collection function collects the edge coverage and stores it in the coverage buffer. *Tardis* can then compare this coverage statistic at the host side with the global coverage bitmap to identify those test cases that trigger new coverage and to guide further test case generation.

A. Coverage Collection

We utilize existing compiler infrastructure to conduct OS-agnostic code coverage collection as well as guarantee the portability of the proposed mechanism. Specifically, the Clang compiler supports a general instrumentation mechanism that can inject user-defined callbacks at the beginning of each code block. The callback *pc_trace_guard_init()* is used to initialize the whole collection procedure and is called before entering program logic, while the callback *pc_trace_guard()* is invoked at the entry site of each basic block for code branch collection. Based on the aforementioned mechanism, we can implement Embedded OS coverage collection. Since such a mechanism of Clang depends only on the source code information, we can guarantee the portability and independence of the proposed coverage collection. However, the callback used for initialization is designed for user land programs, and the resource for running the Embedded OS is limited, thus we need to design a dynamic initialization mechanism and efficient coverage collection callback.

Dynamic Initialization. As mentioned in Section II, the variety of compilation environments and the diverse implementation strategies lead to a diverse runtime model and inadequate compilation supports. As a consequence, the existing mechanism for coverage collection initialization provided by Clang compiler may not get invoked during the target kernel booting phase. Specifically, to enable the coverage collection, two types of callbacks functions need to instrument into the OS code. In detail, we utilize the *pc_trace_guard()* to collect edge coverage, while it requires the *pc_trace_guard_init()* to initialize the coverage, i.e., the *pc_trace_guard_init()* assigns ID numbers to each basic block for identification. These functions need to be inserted into certain positions, e.g., the entry of each basic block or before the program's major logic. For a user land program, leveraging its unified runtime model and comprehensive compile support, the *pc_trace_guard_init()* function will get invoked automatically at the program startup. The callback *pc_trace_guard_init()* is invoked by a special instrumented function *__sanitizer_cov_module_init()*, which is called before entering each module of the program so that the collection can be initialized correctly. However, most Embedded OSs adopt highly customized building and linking procedures to produce target binary, where the runtime model is totally different from user land programs, making the callback mentioned for initialization work improperly. Therefore, we propose a dynamic initialization mechanism, independent of both target architecture and target kernel, to address the aforementioned issue.

To load the *pc_trace_guard_init()* properly, we design the dynamic initialization mechanism that can automatically invoke it at the Embedded OS's startup phase. The overall procedures it is shown in Figure 5. In detail, the target OS's source code will be compiled into an ELF image. Then during the load time, for a function that needs to be invoked, we need to pinpoint the exact position of the function. We can then implement an indirect call to it during OS startup through a function pointer. In order to achieve that, we operate based on the binary image of the target OS. This is because such a function is often linked through a library and is not visible to the OS before the linking stage, so we can not implement direct calls to the function at the OS source code level. Concretely, since the initialization function is loaded into the OS code at the linking time, it must already reside at a certain address of the OS address space. Therefore, to acquire the accurate position, we disassemble the entire OS image after the compilation and then find the corresponding address of the target function *pc_trace_guard_init()*. In this way, we can properly call it at OS boot time by means of function pointers.

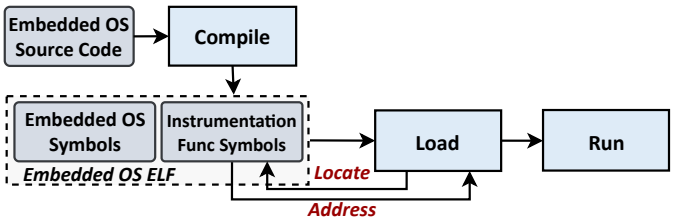


Fig. 5. Diagram of the initialization procedures. The Embedded OS source code will be compiled into an elf image, which contains the target OS's symbols as well as the instrumented function's symbol. During execution, OS will try to locate the address of the instrumented function and call it through the function pointer.

Also, since *pc_trace_guard_init()* is in charge of the coverage initialization, it will iterate every basic block within the target OS. In other words, when invoking the *pc_trace_guard_init()*, we need to provide it with arguments like *start* and *stop*, which in detail, are the addresses of the first basic block and the last basic block of the whole elf. It is worth noting that, different from the *pc_trace_guard_init()* that are loaded at the linking stage, values like *start* and *end* are assigned by Clang directly on the OS space. Therefore, we can directly access them via pointer dereferencing.

OS Agnostic Coverage Collection. Different Embedded OSs normally provide different interfaces to user land programs. For instance, the prototype of most system calls in *UC/OS* and *FreeRTOS* are totally different. In order to apply the coverage collection to a wide range of Embedded OSs, we need to propose an OS-agnostic coverage collection mechanism. Existing coverage collection tools, e.g., *KCOV*, utilizing sophistical services from target OS, e.g., *mmap, fopen,* etc., making them unavailable in embedded scenarios. Therefore, we propose to collect coverage statistics with several simple binary instruments like *bitwise* in an OS-agnostic manner.

As we mentioned above, once we finish the initialization, we can know which blocks have been hit by a test case during the execution, through invoking the *pc_trace_guard()* at each basic block. To calculate this hit count, we first need to identify which edges a test case covers. However, we can not give each edge an ID number directly since the edge is only an abstraction of the program's control flow. Intuitively, we can give each edge a hash number based on its previous and current basic blocks. Nevertheless, such an approach may face several issues. For instance, a control flow usually contains direction information, while the hash operation does not consider the order of the two parameters before the calculation, and this may cause us to neglect the

edge's direction during the collection. Such a problem may cause us to fail to portray the program's coverage accurately. Moreover, hash operations involve many arithmetic operations. Such an extensive arithmetic operation can significantly affect the fuzzing efficiency.

In order to address the aforementioned issues, we utilize the ID assigned to each basic block. By using the *xor* operation on the previously covered basic block and the current basic block, we can have a unique ID for each edge. Meanwhile, to entitle the direction information to each edge, we will *shift* the previous basic block ID by one bit to the right after execution so that we can distinguish between onward and backward control flow. In this way, we can index such an edge directly on our coverage bitmap and statistics its hit counts. Such an approach only uses highly efficient binary operations, which lowers the calculation overhead. Also, the *xor* operation and the *shift* can take the direction of control flow into consideration, enabling an accurate coverage statistic.

B. Coverage Analysis

The coverage analysis phase of fuzzers needs to be efficient enough since such analysis is hot path of the whole fuzzing campaign and is conducted after every time a test case is executed. Embedded OSs normally runs in a hardware environment with limited memory space and computing resources, thus we propose to conduct coverage analysis directly at the host side. However, the coverage statistics are collected inside the target kernel that is executed by a QEMU virtual machine, prohibiting fuzzer from accessing the collected coverage. In order to conduct coverage analysis, we propose to directly access the coverage storage via exposing partial memory space of QEMU so that the overhead of data transferring to the host can be eliminated, and perform analysis with a cpu cache friendly mechanism to improve the efficiency.

Host-VM Coverage Buffer. To conduct a high efficient coverage guided fuzzing, the first and foremost is to have a data buffer that can facilitate the coverage collection, transmission, and analysis. However, coverage analysis tends to be computationally intensive since it needs to inspect the entire program's execution trace after each system call is executed. Suppose we apply coverage analysis directly to the executor with only limited resources. In that case, it may take up much time leading to a poor fuzzing effect. Moreover, since Embedded OS usually have different implementations and are deployed at different architecture. It is hard for them to have a unified and efficient interface to perform the data transfer.

To this end, *Tardis* proposes using a shared coverage buffer to store the coverage information so that it can be accessed on both the guest side and host side directly. Intuitively, QEMU contains the entire memory space of the target Embedded OS. Suppose we can access certain positions of QEMU's memory space, we can then access the corresponding position of the target Embedded OS directly on the host side. However, the modern operating system usually has a certain protection mechanism that each process is isolated from each other, i.e., different processes can not access each other's memory space. That is to say, QEMU's memory space is only visible to itself, and other processes like the fuzzer process do not have permission to access it.

To overcome this boundary, we propose to expose certain data fields within QEMU, like the coverage buffer, using the shared memory mechanism. So that during the fuzzing, we can directly read and write the data buffer that stores the coverage information from the host. In detail, we can locate the address where the instrumented function stores the coverage information. Then, we can modify the QEMU, allowing it to expose such memory segments during the QEMU booting phase. To be specific, during the QEMU booting phase, it will dynamically allocate memory space for the guest OS. When the QEMU attempts to allocate memory for the coverage buffer, we will intercept such operation and try to expose this buffer using the shared memory. By doing so, we can have permission to access this coverage buffer at the host and read the coverage information in a target Ignorant way.

Algorithm 1: Coverage Collection at Host.

```
   Input: QemuShareMemoryPath
1  EmptyBuffer := ∅
2  SharedBuffer := ∅
3  mmap(QemuShareMemoryPath, SharedBuffer)
4  for Call ∈ Prog do
5      // Executor in the guest sets execution state into S
6      S := ReadExecuteStatus(SharedBuffer)
7      if State == WAIT then
8          R := ReadCover(SharedBuffer)
9          SharedBuffer = ResetCover(EmptyBuffer)
10         S := READY
11         SharedBuffer = SentStatueToVm(S)
12     end
13 end
```

Concretely, Algorithm 1 shows how the shared data buffer is accessed during fuzzing on the host side. To begin with, it utilizes the share memory path as input. Then, *Tardis* will map the shared memory, where resides the guest coverage information in the fuzzer, as shown in line 3. Furthermore, *Tardis* collects the coverage statistic on each system call. Before each system call is executed on the target OS, *Tardis* will check whether the OS has finished its last execution, as shown in lines 6-7. Once we know it is waiting for the next system call, we will read the last system call's coverage, then reset the corresponding data buffer, as shown in lines 8-9. Last, we can reset the execution status and tell the fuzzer we are ready for further execution, as shown in lines 10-11. It is noteworthy that, since such a method is based on QEMU only, it is completely OS independent and OS transparent.

Efficient Analysis in Host. After collecting the coverage information and storing them in a shared buffer, we can now analyze them at the host to identify whether the last execution covers any new path. To achieve that, we need to compare the current coverage statistic with an overall coverage statistic. However, coverage comparison is a frequent operation that must be performed after each input's execution. Such a high-frequency and complex operation can take up much time and

significantly affect the fuzzing efficiency. Furthermore, there are certain repetitive operations during program execution, e.g., recursion or iteration. Such operations can lead to huge hit counts for a particular edge, which will affect our accuracy of a new path and thus reduce efficiency. Therefore, to reduce the overhead and boost fuzzing efficiency, we propose to utilize a compact bitmap to record the global coverage and check for new coverage with several simple binary operations.

First, we utilize a compact global coverage bitmap with the size of 64KB, thus allowing the map to be analyzed in a matter of microseconds on the receiving end, and to effortlessly fit within the L2 cache of the host CPU. To perform binary operations, we propose to use a classification filter to convert the edge coverage into binary form. We make a simple hash map of our hit count to facilitate subsequent binary operations. Specifically, since we are using a *char* size to store the hit count of each edge, the hit count of all edges will not exceed 256 times. Furthermore, each char has 8 bits, and we will categorize the hit count into bit level. Finally, when we obtain the coverage bitmap after execution, we compare it with the global coverage bitmap through binary operations. Assuming that the result is not equal to zero, we consider that a test case triggers new coverage. Eventually, the coverage bitmap will be merged with the global coverage bitmap. Since the aforementioned coverage storage is designed to be cache friendly and the whole comparison only involves simple binary operations, the efficiency of the analysis can be guaranteed.

IV. Implementation

Overall, we implemented *Tardis* using Rust for the host fuzzer and C for the guest executor programs. *Tardis* can conduct fuzz testing on *UC/OS*, *FreeRTOS*, *RT-Thread* and *Zephyr* with slight modifications each. Specifically, to adapt to different Embedded OSs, the integration overhead can be divided into compiling the system call descriptions, adapting the host fuzzer and the guest executor programs. To compile the system call descriptions of a given Embedded Operating System, we manually construct these descriptions through reading its API references and convert them to the corresponding description language. The host fuzzer needs to specify the target OS's architecture and the location at the fuzzer's initialization. The guest executor is implemented as a running application in Embedded OS, which is in charge of OS initialization, test case deserialization and execution. The deserialization and execution procedures only consist of arithmetic and memory operations and are devoid of any OS-dependent APIs, thus they can be directly used across different Embedded Operating Systems. While the OS initialization involves many OS-dependent operations as the internal API for these operations is different due to the difference between the OS' implementations.

In order to conduct the coverage guided fuzzing effectively, *Tardis* leverages the architecture shown in Figure 6. Overall, *Tardis* works on both the host side and guest side. On the host side, the primary responsibility is to manage the whole fuzzing process as well as the test case generation. Concretely, for the test case generation, it parses the predefined system call descriptions and generates the initial corpus. Then, during the execution, based on the feedback information such as coverage and crashes, it saves interesting test cases into the corpus and mutates them in later execution. Last, it serializes a test case into a plaint structure to facilitate the data transfer. For fuzzing process management, *Tardis* boots QEMU on each fuzzer startup and reboot the QEMU after each crash. Also, *Tardis* will log all information asynchronously. It monitors and collects valuable information, e.g., the crash information, the coverage information, from multiple QEMU instances. Then it logs them on the host machine during the entire fuzzing campaign. To communicate the host and the guest, we slightly modified QEMU, using the shared memory mechanism to mmap certain QEMU's memory space, so that QEMU can directly expose its internal space, where the target OS's memory locates. Hence, we can directly access the OS's memory space from the host side. Any data I/O like writing test cases or reading coverage information, would be transparent to the target OS. On the guest side, once *Tardis* finishes the OS initialization, it mainly deals with received test cases. Specifically, it deserializes the test cases, extracts the system call and its parameters, and executes them on the target OS via function pointers. The instrumented function will record the coverage information and send them back to the host.

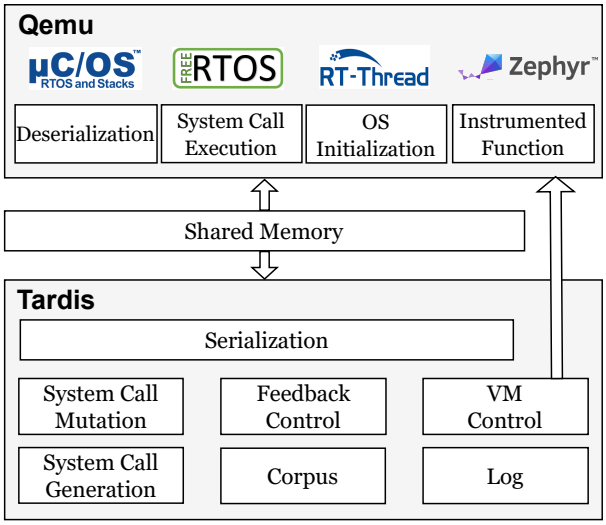


Fig. 6. Diagram of *Tardis* architecture. First, the host side is in charge of the test case generation, mutation, serialization, feedback analysis, record operations, and VM control. Then, the host and guest communicate through the shared memory pass data like coverage and test cases. Last, the guest side is in charge of OS initialization, test case deserialization and execution, normally, there are multiple QEMU instances for each fuzzing process, each of which has a corresponding shared memory region exposed by QEMU.

The instrumentation mechanism is implemented upon Clang's *SanitizerCoverage*. In detail, we reserve a buffer to store the coverage information. Then we design an operating system independent coverage collection mechanism. It contains a coverage initialization function that can iterate every basic block of the target OS and assign them with an ID number. Also, it has a coverage collection function that can collect the coverage information during runtime.

V. Evaluation

We list the following research questions to help us understand *Tardis*'s performance and effectiveness.

- **RQ1** Is *Tardis* able to uncover new bug in different Embedded OSs?
- **RQ2** Is *Tardis*'s coverage guided mechanism effective in term of achieve higher code coverage, comparing with the black box fuzzing?
- **RQ3** What is the overhead that the instrumentation brings to *Tardis* during fuzz testing?

To determine whether *Tardis* is capable of uncovering new bugs and answer **RQ1**, we used *Tardis* to test four open sourced Embedded OS implementations. Overall, *Tardis* found 17 previously unknown bugs. To answer the **RQ2**, we implemented a *Tardis*- with the coverage guidance disabled, to evaluate the effectiveness of the coverage guided generation mechanism. To measure the performance impact of our design and answer **RQ3**, we analyzed how much memory growth instrumentation brings to Embedded OS and how much execution overhead it introduces during the fuzzing campaign.

A. Experiment Setup

We evaluated *Tardis* against the following four Embedded OSs: UC/OS [13], FreeRTOS [2], Rt-Thread [33], and Zephyr [15]. Specifically, we further test UC/OS and FreeRTOS with two different versions: UC/OS-2, UC/OS-3, FreeRTOS-LTS (Long Term Support Version), and FreeRTOS-10.4 (Latest Version). Also, to adapt fuzzing to the above targets, we predefined some system call descriptions based on their user manuals and code specifications. These descriptions are mainly for their core modules, e.g., kernel module, timer module, and some essential peripheral modules like networks. For coverage evaluation, to verify the effectiveness of coverage guidance as well as eliminate the potential impact from implementation details, we implemented *Tardis*-, which in turn is the *Tardis* without the coverage guidance. For the overhead evaluation, we prepared OS images before and after instrumentation to evaluate the memory overhead by comparing the size of the images. We also count the time to execute those inputs that trigger the new coverage before and after instrumentation to evaluate the execution overhead.

The experiments were conducted on a Linux server with 64 GB of memory and a 16-core CPU. Each kernel is compiled by Clang with instrumentation related configuration enabled. We configure each virtual machine instance for every single experiment with the same parameters. Specifically, each fuzzer instance has 2 VM instances with 4 GB of memory and two processor cores. Each experiment is repeated three times, with a duration of 24 hours.

B. Bug Detection Capability

To evaluate *Tardis*'s bug detection capabilities in different Embedded OSs and answer the **RQ1**, we collected and analyzed the crashes reported by *Tardis* during the fuzzing campaigns. Specifically, *Tardis* found a total of 17 previously unknown bugs, as listed in Table I. A more specific breakdown is as follows.

TABLE I
Reported bugs found by *Tardis*

kernel	operations	bug type
UC/OS-3	NetUtil_16BitSumDataCalc	logic error
UC/OS-3	NetSock_BindHandler	logic error
UC/OS-3	NetSock_TxDataHandlerDatagram	logic error
UC/OS-3	NetTCP_RxPktConnHandlerFinWait2	logic error
UC/OS-3	NetNDP_NextHop	logic error
UC/OS-3	NetMLDP_HostGrpLeaveHandler	logic error
UC/OS-3	NetMLDP_HostGrpLeave	logic error
UC/OS-3	Mem_Copy	logic error
UC/OS-2	NetIF_TxIxDataGet	logic error
UC/OS-2	NetIGMP_TxMsgReport	logic error
FreeRTOS-10.4	xAreTimerDemoTasksStillRunning	logic error
FreeRTOS-10.4	xTaskCheckForTimeOut	null-ptr deref
FreeRTOS-10.4	xSecondTimerHandler	logic error
FreeRTOS-LTS	xTaskResumeAll	null-ptr deref
RT-Thread	lwip_getaddrname	null-ptr deref
RT-Thread	rt_spi_sendrecv8	logic error
RT-Thread	rt_vsnprintf	null-ptr deref

In total, we found 4 vulnerabilities on FreeRTOS, 10 vulnerabilities on UC/OS, and 3 vulnerabilities on RT-Thread. By far, 1 vulnerability has been fixed by the maintainer. We find that most of these vulnerabilities are located in the target OS's kernel module and network-relevant modules since our system call descriptions specifically target these sections. We further analyze the potential effect of these vulnerabilities through the help of the GDB debugger. Results indicate that one of them can trigger a memory fault, while the rest can cause the system to become unresponsive or crash. *Tardis*'s finding these vulnerabilities is a direct result of the coverage guidance mechanism since it enables *Tardis* to identify those interesting test cases and give them a higher chance of mutation. In this way, we can keep generating high quality test cases, thereby testing deeper into the Embedded OS's code logic, uncovering those previously unknown vulnerabilities.

Figure 7 shows a previously unknown vulnerability in the internal function *NetMLDP_HostGrpLeave()* of the *UC/OS*. The vulnerability was found by *Tardis* during the fuzzing campaign and was confirmed and fixed by maintainers from UC/OS community. As shown in the figure, the variable *host_grp_leave* is declared at the beginning of the function *NetMLDP_HostGrpLeave()* (line 4) without a proper initialization. After successfully acquiring the global lock used in the network subsystem (line 5 to line 8), the execution path would jump to the label *exit_release* if the value of the input pointer *p_addr* equals to zero (line 13 to line 16), which means the caller does not provide the needed buffer for a network address. The whole process would return at the jumped site *exit_release* with the variable *host_grp_leave* that contains the garbage value returned to the caller (line 23 to line 26), resulting in the undefined behavior. The returned garbage value can result in unexpected behavior at the caller site, further leading to various critical security issues.

Although the *UC/OS* has been widely deployed for a prolonged time and *UC/OS* itself contains a rich set of unit tests as well as system level tests, none of them can discover and report the aforementioned vulnerability. With the proposed coverage collection mechanism and the corresponding guided fuzzing procedure, *Tardis* is capable of retaining valuable test

```
1  int NetMLDP_HostGrpLeave (NET_IF_NBR if_nbr,
       NET_IPv6_ADDR *p_addr, NET_ERR *p_err)
2  {
3  // host_grp_leave is declared without
       initialization.
4      CPU_BOOLEAN host_grp_leave;
5      Net_GlobalLockAcquire((void
           *)&NetMLDP_HostGrpLeave, p_err);
6      if (*p_err != NET_ERR_NONE) {
7          goto exit_lock_fault;
8      }
9      #if (NET_ERR_CFG_ARG_CHK_EXT_EN ==
           DEF_ENABLED)
10
11     #if (NET_ERR_CFG_ARG_CHK_EXT_EN ==
           DEF_ENABLED)
12 // if p_addr is equal to NULL
13     if (p_addr == (NET_IPv6_ADDR *)0) {
14         NET_CTR_ERR_INC(Net_ErrCtrs.NullPtrCtr);
15         *p_err = NET_ERR_FAULT_NULL_PTR;
16         goto exit_release;
17     }
18     #endif
19     host_grp_leave =
           NetMLDP_HostGrpLeaveHandler(if_nbr,
           p_addr, p_err);
20         goto exit_release;
21 exit_lock_fault:
22         return (DEF_FAIL);
23 exit_release:
24         Net_GlobalLockRelease();
25 // Undefined or garbage value returned to caller
26         return (host_grp_leave);
27     }
```

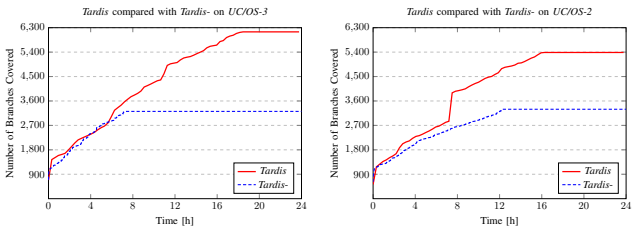
Fig. 7. A previously unknown vulnerability in the *UC/OS* kernel was found by *Tardis* during the fuzzing campaign. The variable *host_grp_leave* is declared at line 4 without initialization, which leaves a garbage value in the corresponding memory location. After acquired the global lock successfully (line 5 to line 8), the execution would jump to the label *exit_release* if the input pointer *p_addr* is a null pointer (line 13 to line 17), where the uninitialized variable *host_grp_leave* is returned without any further handling procedure (line 23 to line 26), thus transferring the garbage value to caller.

cases that cover the new execution path for further mutations, thereby improving the quality of the generated test cases. *Tardis* can find such security issues in a limited time, which demonstrates its bug finding capability.

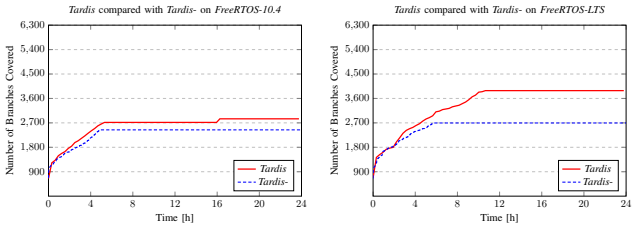
C. Coverage Statistics

We here to address **RQ3** and evaluate whether the coverage guidance can assist *Tardis* in achieving a better code coverage. Note that *Syzkaller* currently does not nativity support fuzzing Embedded OSs, since it requires many features from *posix* platform, e.g., spawning threads dynamically, and needs kernel features like *KCOV*, which Embedded OS does not provide. Hence, instead of comparing the effectiveness with tools like *Syzkaller*, we evaluate the effectiveness of the coverage guidance mechanism.

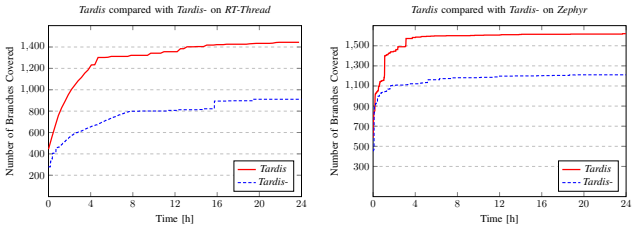
In detail, to evaluate the effectiveness of the coverage guidance mechanism, we designed *Tardis-*, which in turn is the *Tardis* with the coverage guidance functions disabled. The *Tardis-* can only collect the coverage statistic, but do not use the feedback as guidance, for example, it can not identify and preserve interesting test cases.



(a) Branch Coverage Statistics on UC/OS



(b) Branch Coverage Statistics on FreeRTOS



(c) Branch Coverage Statistics on RT-Thread and Zephyr

Fig. 8. Coverage statistics of *Tardis* with and without (*Tardis-*) coverage feedback on UC/OS-3, UC/OS-2, FreeRTOS-10.4, FreeRTOS-LTS, RT-Thread, and Zephyr over 24 hours.

TABLE II
BRANCH COVERAGE STATISTICS OF *Tardis* WITH AND WITHOUT (*Tardis-*) COVERAGE GUIDANCE ON THE RESPECTIVE EMBEDDED KERNELS. FOR DIFFERENT EMBEDDED OSES, FRS MEANS FREERTOS, RTTRD MEANS RT-THREAD, AND ZPHR MEANS ZEPHYR.

	UCOS-3	**UCOS-2**	**FRS-10.4**	**FRS-LTS**	**RTTRD**	**ZPHR**
Tardis	6146	5387	2849	3888	1444	1620
Tardis-	3216	3295	2442	2696	910	1211
Avg-impr	91.11%	63.49%	16.67%	44.21%	58.68%	33.77%

The coverage results are demonstrated in Table II. Comparing with *Tardis-*, *Tardis* has a coverage increment about 91.11%, 63.49%, 16.67%, 44.21%, 58.68%, and 33.77% in UC/OS-3, UC/OS-2, FreeRTOS-10.4, FreeRTOS-LTS, RT-Thread, and Zephyr respectively. On average, *Tardis* gains a 51.32% of coverage improvement. As we can infer, the coverage guidance mechanism greatly facilitates the fuzzing performance. The reason behind this is that, compared to black-box fuzzing, *Tardis* adapts to the coverage guidance mechanism, which helps it locate and discover test cases that trigger new coverage. These high-quality test cases are given a higher probability of mutation, and such extensive mutations on them improve the quality of newly generated test cases, allowing *Tardis* to reach deeper code logic than *Tardis-* does.

Based on the sampling data, we further plot the coverage growth curve as shown in Figure 8. We notice that *Tardis* can reach a higher code coverage at a faster speed with the help

of coverage guidance compared to black box fuzzing. Specifically, taking FreeRTOS-10.4 as an example, *Tardis* can grow at a faster rate than *Tardis*-, and it can grow at a longer time and finally reach a higher coverage. The fast growth mainly contributes to the coverage guidance mechanism, which helps the fuzzer identify valuable test cases, thereby gaining a rapid growth rate. It is worth noting that at some point, the coverage data may suddenly increase, as is the case in UC/OS-2 when *Tardis* is running around 8 hours. In this case, the fuzzer may generate test cases for some untested modules. Since these modules have never been covered, these test cases may be able to go very deep into this module, and then there is a possibility of a sudden increase in coverage.

Also, we notice that both *Tardis* and *Tardis*- are prone to reach the saturation status, where the coverage barely grows, e.g., in FreeRTOS-LTS, both *Tardis* and *Tardis*- have no more growth after 12 hours. This is because writing these test cases descriptions requires considerable manual effort, which is another topic that the primary purpose of this paper does not attempt to address. Therefore, we only provide descriptions for some core modules with limited test cases. This could lead to their being tested rapidly, making the fuzzing campaign easy to saturation. A more comprehensive description generation technique can alleviate this issue and let *Tardis* cover significantly more branches.

D. Instrumentation Overhead

Memory Usage. Given that fuzzing campaigns are typically performed in parallel, the memory consumption of the target is essential. Since Embedded OSs are designed for scenarios with limited memory resources. Therefore, to minimize the high memory overhead that affects our fuzzing efficiency, we need to reduce the memory space occupied by instrumentation as much as possible. Here we compare the memory growth of the target Embedded OS before and after the instrumentation.

TABLE III
MEMORY CONSUMPTION OF INSTRUMENTATION DURING FUZZING. FOR DIFFERENT EMBEDDED OSES, FRS MEANS FREERTOS, RTTRD MEANS RT-THREAD, AND ZPHR MEANS ZEPHYR.

	UCOS-3	UCOS-2	FRS-10.4	FRS-LTS	RTTRD	ZPHR
Uninstrumented	1853Kb	1929Kb	419Kb	424Kb	2330K	531Kb
Instrumented	2332Kb	2219Kb	650Kb	684Kb	2971K	799Kb
Overhead	20.54%	13.07%	35.54%	38.01%	21.58%	33.53%

Table III shows the detailed memory growth brought by the instrumentation. As we can infer from the table. The UC/OS increased 20.54% and 13.07% for UC/OS-3 and UC/OS-2. For FreeRTOS, the memory consumption grows about 35.54% and 38.01% in 10.4 and LTS versions. While the RT-Thread and Zephyr increases about 21.58% and 33.53% speratly.

The memory increment of all four Embedded OS is 27.05% of increment on average. Such memory growth is inevitable because instrumentation requires inserting coverage collection functions and coverage buffers into the target Embedded OS. Also, we note that UC/OS has a lower memory growth rate compared to FreeRTOS. This is because we are currently only instrument C code, despite that UC/OS has a larger code base than FreeRTOS, it contains more non-instrument code, such as assembling. Overall, the memory overhead caused by instrumentation is acceptable. This overhead does not contribute significantly to the overall memory usage, and the impact of this consumption on the overall fuzzing performance can be negligible.

Execution Time Overhead. We further evaluate the execution overhead imposed by instrumentation. Specifically, execution overhead analysis usually requires the support of some system libraries, while in the bare-metal scenario the lack of such necessary support makes it difficult to perform direct overhead analysis. Here we choose to count the overall time that the target Embedded OS executes all interesting corpus as a comparison metric. Specifically, we collect those test cases that trigger new coverage during fuzzing as a dataset and re-execute them on instrumented and uninstrumented Embedded OS, statistics and compare their respective execution times.

TABLE IV
TIME OVERHEAD OF INSTRUMENTATION DURING FUZZING. FOR DIFFERENT EMBEDDED OSES, FRS MEANS FREERTOS, RTTRD MEANS RT-THREAD, AND ZPHR MEANS ZEPHYR.

	UCOS-3	UCOS-2	FRS-10.4	FRS-LTS	RTTRD	ZPHR
Uninstrumented	4395.20s	4175.44s	25517.03s	26511.20s	688.56s	219.78s
Instrumented	5933.52s	5713.76s	29825.10s	33139.00s	918.08s	299.70s
Overhead	35.0%	36.7%	16.9%	25.0%	33.3%	36.4%

Specifically, we have selected a corpus of 21976, 66278, 5738, and 1998 test cases of interest for UC/OS, FreeRTOS, RT-Thread, and Zephyr. Here we will count the execution time of the corresponding corpus for each Embedded OS respectively. Table IV shows the detailed time increase caused by the instrumentation. We can see from the table that instrumentation leads to an average execution overhead of 30.55%. For the six Embedded OSs, it increases the execution overhead by 35.0%, 36.7%, 16.9%, 25.0%, 33.3%, and 36.4%, respectively. Such overhead is caused by the fact that we insert coverage processing functions that require frequent read and write operations on the coverage buffer before each basic block of the target Embedded OS. Although the coverage guided mechanism of *Tardis* now introduces some overhead, the impact of this overhead on the performance of the coverage fuzzing processing activities is negligible compared to the benefits it brings.

VI. DISCUSSION

Extensibility. Currently, *Tardis* is mainly written in Rust, while its executor is written using C, and we use QEMU to support Embedded OS emulation. This framework avoids the problems caused by the different Embedded OS targets as much as possible. However, *Tardis*'s performance may have some limitations due to the following two aspects. On the one hand, the QEMU-based fuzzing may have a relatively slower speed than real hardware testing. Also, although many kernel fuzzers like Syzkaller apply QEMU-based fuzzing, deploying OSes, especially embedded OSes that supporting dozens of architectures on emulators, there may be certain hardware emulation like peripheral devices that is not supported on some architectures. In this case, QEMU can not simulate them,

which in turn leads to this part of code not being tested. On the other hand, *Tardis* uses system call descriptions as input, just like any other OS fuzzers, which contain detailed information about the system calls, including return values, parameters, etc. These system call descriptions are highly structured and have high requirements for their accuracy. Since incorrect system call descriptions make it impossible to test the deep logic of Embedded OS. However, such descriptions can only be written manually and place high demands on the developer. Because writing such descriptions need to have a certain level of understanding of the Embedded OS to be tested, which undoubtedly increases the difficulty of our adaptation. To adapt more Embedded OS, we can perform dynamic analysis through OS tracing tools, like *ptrace*, to collect the real execution traces and derive the detailed semantic information of a system call like parameter types and values. Alternatively, we can directly analyze the parameter types and relationships of the system call through static analysis. Thus automate generating such system call descriptions and reduce our adaptation cost.

Bug Detection Capability. Although by means of coverage guidance, we are now able to detect 17 bugs. We mainly detect Embedded OS those unrecoverable errors by the generated input. Specifically, an Embedded OS will perform frequent checks on its internal execution state during runtime. If it finds that the values of certain parameters are in an unrecoverable state, Embedded OS will enter the error handling code and trigger a system hang eventually. For now, we can only determine whether there are problems with the system when a hang is detected. Specifically, there are many silent data errors within Embedded OS. For example, many use-after-free errors do not affect the overall execution of the system. However, such hidden errors can lead to severe consequences, such as remote code execution, information leakage, etc. In the subsequent tests, to enhance the bug detection capability of *Tardis*. We can make use of some monitors of QEMU to detect the registers of Embedded OS, or the execution, and thus find these silent errors.

VII. Related Work

Kernel Fuzzing. The operating system kernel is inevitable to have tremendous vulnerabilities within its codebase. Therefore, more and more researchers have attempted to port fuzzing into kernel testing [4], [5], [10]–[12], [17], [24], [30]. For the general purpose operating system fuzzing. *Syzkaller* [28] is a state-of-the-art coverage guided kernel fuzzer developed by Google. It has explored many severe kernel vulnerabilities [27] and can test a wide range of kernels like Linux and Windows. It proposes to use the system call description as input [29], so it can encode rich semantic information into test cases to maximize the fuzzing efficiency. There are works that try to boost the fuzzing speed by various methods. For example, Healer [26] tries to generate high-quality test cases by uncovering the hidden relations between two adjacent system calls. Also, some works try to combine symbolic execution and kernel fuzzing. Specifically, symbolic execution utilize path constraints to calculate the symbolic values to test programs. For example, HFL [11] adopted symbolic execution to generate high-quality input for those hard-to-pass checkpoints in kernel. Also, by limiting the range of analysis paths, HFL greatly avoids the huge computation overhead that symbolic execution incurs during kernel analyzing.

The above works are mainly focused on the GPOS fuzzing; they may find it challenging to port it in Embedded OS fuzzing. Therefore, some works try to extend fuzzing into other operating system types. For example, *Gustave* [6] is a Embedded OS fuzzer, it realized a customized QEMU board to emulate the target Embedded OS. Then it uses the QEMU-TCG to achieve the binary instrumentation for coverage guidance and memory-related vulnerabilities detection. KAFL [21] uses Intel-PT to enable the coverage guided fuzzing, which is an vendor-specific hardware feature. Such mechanism is limited to X86 architecture OS only, however, most Embedded OS are deployed on ARM or RISC-V architecture. Also, Rtkaller [24] proposes to use tasks as fuzzing inputs and uses the parallel execution to trigger kernel's scheduling behavior for an efficient RTOS fuzzing. However, it is based on *Syzkaller*, currently only support rt-Linux fuzzing, can not be adapted to other Embedded OS scenarios.

Instrumentation in Fuzzing. Fuzzing [1], [3], [7], [14], [18], [20], [31] is essentially a rather brute-force mechanism that explores the program's vulnerability via random input. This kind of blind testing makes it difficult to understand the internal execution state of the program, which in turn leads to very inefficient testing. Instrumentation [32] is used to give the fuzzer a better internal view of the target program during the fuzzing campaign.

In detail, it usually enables fuzzer with coverage guidance and vulnerability detection by providing edge counts, call stack, and memory access pattern. For the coverage information, different instrumentation tools have different coverage collection granularities. For instance, SanitizerCoverage [19] and AFL-GCC [14] method are the two most widely-used coverage instrumentation tools. SanitizerCoverage offers function level, basic block level, and edge level coverage information, while AFL provides edge level coverage only. Also, the instrumentation helps fuzzer in facilitating vulnerability detection. In detail, The sanitizer toolset can detect a wide range of vulnerabilities in the C/C++ program. Some of the representatives like Address SANitzier (ASAN) [8], [22], can detect any illegal memory access like use after free, double free, and stack overflow. Thread SANitizer (TSAN) [23] and Kernel Concurrency SANitizer (KCSAN) [9] can detect data racing problems, etc.

Main differences. In general, there is a huge difference between general purpose OS fuzzing and Embedded OS fuzzing. So far, most coverage guided kernel fuzzers focus on those kernels with complete infrastructure support, such as Linux, MacOS, and Windows. However, these fuzzers rely heavily on existing infrastructures such as standard APIs and uniform memory layouts. For targets like Embedded OSs, many operations are not allowed due to resource constraints and limited functions support. While there are tools such as *Gustave* and *KAFL* that enable coverage guided fuzzing of certain Embedded OS by heavily modified QEMU and introducing the binary instrumentation. Such an approach often

suffered from a high adaptation cost, and inaccuracies brought by the binary instrumentation itself.

VIII. Conclusion

In this paper, we present *Tardis*, the first coverage guided fuzzer that is able to discover bugs in Embedded OS. *Tardis* proposes a coverage collection mechanism that is able to instrument Embedded OSs and conduct an OS-agnostic coverage collection. The coverage is gathered on-the-fly and stored into a data buffer shared between the host fuzzer and the guest, enabling direct accessing without extra copy. Those inputs trigger new coverage can be detected with an efficient coverage analysis mechanism, thus evolving the whole fuzz campaign. The evaluation shows that the instrumentation brings averagely 27.05% and 30.55% memory consumption and execution overhead. While it gains an improvement of coverage by 51.32% on average, comparing with the black box fuzzing, which demonstrates the effectiveness of the proposed coverage guidance. Furthermore, we found 17 previously unknown bugs among four Embedded OSs, indicating the bug discovery capability of *Tardis*.

References

[1] Domagoj Babić, Stefan Bucur, Yaohui Chen, Franjo Ivančić, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. FUDGE: Fuzz Driver Generation at Scale. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 975–985, New York, NY, USA, 2019. Association for Computing Machinery.

[2] Richard Barry et al. Freertos. *Internet, Oct*, 2008.

[3] Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao, and Zhuo Su. EnFuzz: Ensemble Fuzzing with Seed Synchronization among Diverse Fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1967–1983, Santa Clara, CA, August 2019. USENIX Association.

[4] Jaeseung Choi, Kangsu Kim, Daejin Lee, and Sang Kil Cha. NTFUZZ: Enabling type-aware kernel fuzzing on windows with static binary analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 1973–1989, 2021.

[5] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. Difuze: Interface aware fuzzing for kernel drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2123–2138, New York, NY, USA, 2017. Association for Computing Machinery.

[6] Stéphane Duverger and Anaïs Gantet. Gustave: Fuzz it like it's app. *DMU Cyber Week*, 2021.

[7] Jian Gao, Yiwen Xu, Yu Jiang, Zhe Liu, Wanli Chang, Xun Jiao, and Jiaguang Sun. Em-fuzz: Augmented firmware fuzzing via memory checking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):3420–3432, 2020.

[8] Google. Kernel address sanitizer. https://www.kernel.org/doc/html/latest/dev-tools/kasan.html.

[9] Google. Kernel concurrency sanitizer. https://www.kernel.org/doc/html/latest/dev-tools/kcsan.html.

[10] Dae R. Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. Razzer: Finding Kernel Race Bugs through Fuzzing. In *IEEE Symposium on Security and Privacy*, pages 754–768. IEEE, 2019.

[11] Kyungtae Kim, Dae R. Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. HFL: Hybrid Fuzzing on the Linux Kernel. In *NDSS*, 2020.

[12] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. Finding Semantic Bugs in File Systems with an Extensible Fuzzing Framework. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 147–161, New York, NY, USA, 2019. Association for Computing Machinery.

[13] Silicon Lab. Ucos. https://www.silabs.com/developers/micrium.

[14] lcamtuf. American fuzzy lop, 2013. https://lcamtuf.coredump.cx/afl/.

[15] Anas Nashif. Zephyr is a new generation, scalable, optimized, secure RTOS, 2016. https://github.com/zephyrproject-rtos/zephyr.

[16] Henry Neugass, G Espin, Hidefume Nunoe, Ralph Thomas, and David Wilner. Vxworks: an interactive development environment and real-time kernel for gmicro. In *Eighth TRON Project Symposium*, pages 196–197. IEEE Computer Society, 1991.

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

[18] Gaoning Pan, Xingwei Lin, Xuhong Zhang, Yongkang Jia, Shouling Ji, Chunming Wu, Xinlei Ying, Jiashui Wang, and Yanjun Wu. V-shuttle: Scalable and semantics-aware hypervisor virtual device fuzzing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 2197–2213, New York, NY, USA, 2021. Association for Computing Machinery.

[19] LLVM Project. Llvm sanitizercoverage. https://clang.llvm.org/docs/SanitizerCoverage.htmll.

[20] Sunny Raj, Sumit Kumar Jha, Arvind Ramanathan, and Laura L. Pullum. Testing autonomous cyber-physical systems using fuzzing features from convolutional neural networks: Work-in-progress. In *Proceedings of the Thirteenth ACM International Conference on Embedded Software 2017 Companion*, EMSOFT '17, New York, NY, USA, 2017. Association for Computing Machinery.

[21] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 167–182, Vancouver, BC, August 2017. USENIX Association.

[22] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, page 28, USA, 2012. USENIX Association.

[23] Konstantin Serebryany and Timur Iskhodzhanov. Threadsanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, page 62–71, New York, NY, USA, 2009. Association for Computing Machinery.

[24] Yuheng Shen, Hao Sun, Yu Jiang, Heyuan Shi, Yixiao Yang, and Wanli Chang. Rtkaller: State-Aware Task Generation for RTOS Fuzzing. *ACM Trans. Embed. Comput. Syst.*, 20(5s), sep 2021.

[25] SimonKagstrom. Kcov. https://github.com/SimonKagstrom/kcov.

[26] Hao Sun, Yuheng Shen, Cong Wang, Jianzhong Liu, Yu Jiang, Ting Chen, and Aiguo Cui. *HEALER: Relation Learning Guided Kernel Fuzzing*, page 344–358. Association for Computing Machinery, New York, NY, USA, 2021.

[27] Dmitry Vyukov and Andrey Konovalov. Syzbot, 2015. https://syzkaller.appspot.com/upstream.

[28] Dmitry Vyukov and Andrey Konovalov. Syzkaller: an unsupervised coverage-guided kernel fuzzer, 2015. https://github.com/google/syzkaller.

[29] Dmitry Vyukov and Andrey Konovalov. Syzlang: System Call Description Language, 2015. https://github.com/google/syzkaller/blob/master/docs/syscall_descriptions_syntax.md.

[30] Daimeng Wang, Zheng Zhang, Hang Zhang, Zhiyun Qian, Srikanth V. Krishnamurthy, and Nael Abu-Ghazaleh. SyzVegas: Beating Kernel Fuzzing Odds with Reinforcement Learning. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2741–2758. USENIX Association, August 2021.

[31] Mingzhe Wang, Jie Liang, Yuanliang Chen, Yu Jiang, Xun Jiao, Han Liu, Xibin Zhao, and Jiaguang Sun. SAFL: Increasing and Accelerating Testing Coverage with Symbolic Execution and Guided Fuzzing. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ICSE '18, page 61–64, New York, NY, USA, 2018. Association for Computing Machinery.

[32] Mingzhe Wang, Jie Liang, Chijin Zhou, Yu Jiang, Rui Wang, Chengnian Sun, and Jiaguang Sun. RIFF: Reduced Instruction Footprint for Coverage-Guided Fuzzing. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 147–159. USENIX Association, July 2021.

[33] Bernard Xiong and Man Jianting. RT-Thread is an open source IoT operating system., 2007. https://github.com/RT-Thread/rt-thread.

[34] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. Krace: Data Race Fuzzing for Kernel File Systems. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1643–1660, 2020.