CLC ___________ Number___________

UDC___________ Available for reference □Yes □No

Southern University of Science and Technology

Undergraduate Thesis

Thesis Title: Real-time capturing of system calls on ARM

（中期报告）

Student Name: Haonan Li

Student ID: 11712510

Department: Department of Computer Science and Engineering

Program: Computer Science and Technology

Thesis Advisor: Fengwei Zhang

Date: March 31, 2021

COMMITMENT OF HONESTY

1. I solemnly promise that the paper presented comes from my independent research work under my supervisor's supervision. All statistics and images are real and reliable.
2. Except for the annotated reference, the paper contents no other published work or achievement by person or group. All people making important contributions to the study of the paper have been indicated clearly in the paper.
3. I promise that I did not plagiarize other people's research achievement or forge related data in the process of designing topic and research content.
4. If there is violation of any intellectual property right, I will take legal responsibility myself.

Signature:

Date:

Real-time capturing of system calls on ARM

Haonan Li

(Department of Computer Science and Engineering Advisor: Fengwei Zhang)

[**ABSTRACT**]: Reproducing bugs is difficult. In areas such as application development, developers often encounter errors reported by users. They often have to rely on the report logs uploaded by the user to try to reproduce these failures. Unfortunately, bugs are usually not reproduced such faithfully. This is mainly because the execution of a program is always accompanied by many non-deterministic events. The recording of these non-deterministic events is an effective way to address this issue. Meanwhile, it is noticeable that system calls are the primary source of non-deterministic events, hence we need to capture these system calls.

In this thesis, I developed a system call capturing tool. This tool utilizes Linux Tracepoint to record system calls across the entire system, or filtering by user-determined constraints, with low overhead and transparent trapping.

To demostrate the effectiveness and the efficiency of my research. I evaluate it with real-world bugs and compare it with the state-of-the-art systems.

[**Keywords**]: Linux, Syscall, Record

Contents

1. Intoduction

The program would often fail. To sufficiently understand and prevent failures, developers requires firstly reproduce these bugs, which ensures the same output and bugs no matter how many times it is re-executed. However, directly re-exection is not suitable for non-deterministic failures, as they may not appear in a re-rection procedure. Non-deterministic failures are the consequence of non-deterministic instructions.

Instructions for running a program can be divided into two categories. One is deterministic, i.e., the behavior of the program is determined in each execution. The other type is non-deterministic, meaning that execution in different situations will have different results. Although most of the CPU execution is deterministic, non-deterministic instructions are also pervasive. This is mainly because of the fact that the execution of a program is not in an isolated system. In fact, the operating system plays a critical role in program initialization, system calls, and scheduling throughout the program lifecycle. Typical sources of nondeterminism include system calls, interrupts, signals, and data races for concurrency programs.

All these non-deterministic events can be futher classified into two types: inconstancy of the data flow - for example, certain system calls such as `getrandom` and `getpid`, and inconstancy of the control flow - for example, concurrency bug due to memory access in inconsistent order[1].

Record-and-replay is a type of approaches that addresses this challenge. Most Record-and-replay systems work by first recording non-deterministic events during the original run of a program and then substituting these records during subsequent re-execution. Record-and-replay system could ultimately guarante that each replay will be identical with the initial version. The fact that a number of replay systems have been built and put into use in recent years illustrates the value of record-and-replay systems in practice[2-5].

There is a rich amount of research on record-and-replay systems, and we can find their various treatments of non-deterministic records. Early record-and-replay systems tend to use virtualization techniques so as to observe and record the entire program non-deterministically on the hypervisior, but the virtual machine is very heavy[6, 7]. Some systems use dynamic binary instrumentation to get the results after running each instruction, but this is very inefficient[5]. There are some other systems that choose not to record at runtime in order to address the expensive cost of recording; instead, they infer these non-deterministic events based on the control flow and other information collected.[4, 8]. However, inference often does not reproduce program execution as faithfully as records, and the time required for inference, which in the worst case is a search of the entire space, is a problem[3]. There are also systems that use custom hardware, which inevitably affects its usefulness in practice[9]. Recently there have been some practical systems that have adopted tools provided by Linux for tracing, thus achieving better efficiency. Nevertheless, it still introduces a considerable overhead (50%) and is therefore used in scenarios where the developer exactly needs to debug[2].

This thesis focuses on the data record part of record-and-replay systems, precisely, the logging of non-deterministic events caused by system calls. I argue that a *practical* record system should (1) run online, meaning that the recording has little performance impact on the execution of the target program, (2) log all data without any omission, (3) work on commercial off-the-shelf hardware, (4) not require any modification to the target program, and also (5) not require any modification to the kernel.

In this thesis, I propose ${PROJECT_NAME}, a practical solution for syscall capturing. It works effectively on Linux, on commercial off-the-shelf hardware, and it can run almost imperceptibly along with the original program.

2. Background

2.1 Linux Tracepoint

Tracepoint is a lightweight hook for events provided by Linux. With tracepoint, we can provide a callback function (called a probe) at runtime for Linux to execute when these events are triggered.[10, 11]

There are two events (`sys_enter` and `sys_exit`) defined in `/include/trace/events/syscalls.h` that can be exploited to execute callback function in the procedure of invloking syscall.[10]

2.2 Linux Process Management

Linux uses process descriptors to represent all information about each process, such as the process priority, the address space allocated, the files allowed to be accessed, and so on. Process descriptors are all of type `task_strcut`.[10, 12]

In Linux kernel, there is a macro named `current` defined in `arch/arm64/include/asm/current.h` (on Arm64). `current` is a pointer that refers to the process that is currently executing; precisely, the issuer of the system call.[10, 13]

3. Design

The overarching target of ${PROJECT_NAME} is to capture Linux syscalls with low overhead. ${PROJECT_NAME} realizes syscall capturing in three steps. (1) ${PROJECT_NAME} inspect and record each syscall. (2) Next, ${PROJECT_NAME} filter these records by some arrtibutions of its caller process. (3) Finally, ${PROJECT_NAME} transfer these filtered records.

3.1 Design Choices

When designing ${PROJECT_NAME}, I make three design choices.

- **Modify Kernel Source vs. Load Kernel Module:** Modifying Kernel source may make all procedures much more easy. However, this approach will inevitably reduce the compatibility of ${PROJECT_NAME}, especially for devices whose kernels have been modified by manufacturers. Futhermore, to modify kernel, we would also need to synchronize upstream changes. Consequently, ${PROJECT_NAME} is baesd on kernel moudle and works as a loadbale driver.

- **Source vs. Binary:** I choose to do the capturing at the binary level instead of at the source code level. because it is quite common that source code and system calls have not corresponded (e.g., library function `malloc` do not necessarily call the `mmap`), with most programs use libraries instead of calling system calls directly. In fact, ${PROJECT_NAME} only concerns the runtime of processes, and would not perform any static analysis.

3.2 Challenges

To achieve its desgin goal, ${PROJECT_NAME} faces following challenges.

3.2.1 Syscall parameters

Considering some syscalls that allow to pass pointer and then modify its memory addressed by this pointer, it is necessary to check and record the change. For example, the prototype of syscall `getrandom` is defined as follow:[1]

```
ssize_t getrandom(void *buf, size_t buflen, unsigned int flags);
```

It is clear that the `getrandom` will fill the buffer pointed to by its first parameter, `buf`. Therefore, ${PROJECT_NAME} needs to record the memory area changed by syscall. However, there is still a challenge that sometimes we cannot get the address at the end of syscall involked. This is due to the way the parameters are passed, determined by the architecture of CPU.

In Arm64[14], both first argument(`arg1`) and return value are saved in register `r0`, which means the parameter stored in `r0` will be replaced with return value before the syscall return to user. This results in the inability to directly record the chanegs only at the end of syscall handled in kernel. I solve this challenge by adding an extra record for several syscalls.

3.2.2 Record Transfer

System call records need to be stored in a buffer, while it is a very important issue when it comes to somehow get them from the buffer in real-time, with low consumption, and

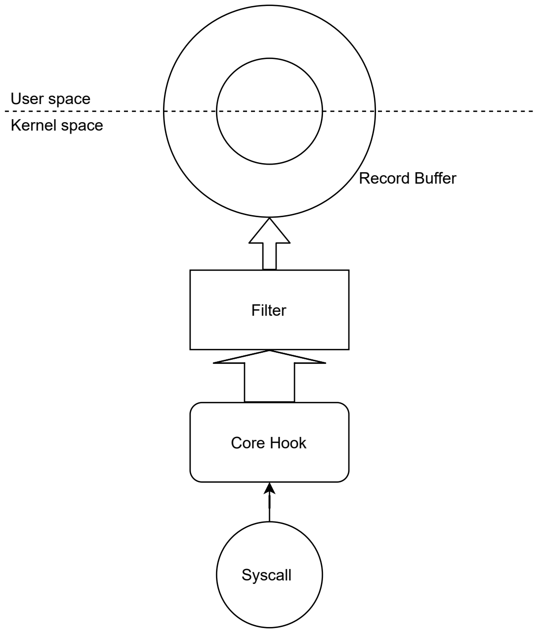

Figure 1: The three parts of ${PROJECT_NAME}

completely. However, the most strightforward solution, i.e., trying to keep fetching data from the buffer and save to file, causes a huge amount of overhead. I solve this challenge by enlarging buffer and reducing the frequency of acquisition.

In this section, I present the desgin of ${PROJECT_NAME} by focusing on how it addresses the above three key challenges. ${PROJECT_NAME} contians three parts: *core hook*, *filter*, and *record transfer*.

As Figure 1 shows, in the kernel space, *core hook* will firslty inspect each syscall and then transfer relevant information to *filter* part. Subsequently, at the *filter* part, syscall records with specific features (e.g., process id or name) will be selected and finally pass to *record transfer*. Meanwhile, a daemon will check the buffer periodically and dump these data form buffer to file.

3.3 Core Hook

Figure 2 shows the general workflow of *core hook*. It consists of two stages of hooks, at the beginning and end of kernel handling of syscall. For the stage 1, ${PROJECT_NAME} will follow the start of syscall handlers and save the value of first parameter, if this syscall may change the memory addressed by first argument.

The second stage takes on more responsibility, including the recording of other pointer type parameters (except for the first one), and return values. Besides, this stage should also get the relevant information collected by stage 1.

There is still a problem that, due to the concurrency of the system, there may be multiple system calls being processed at the same time. Especially considering that

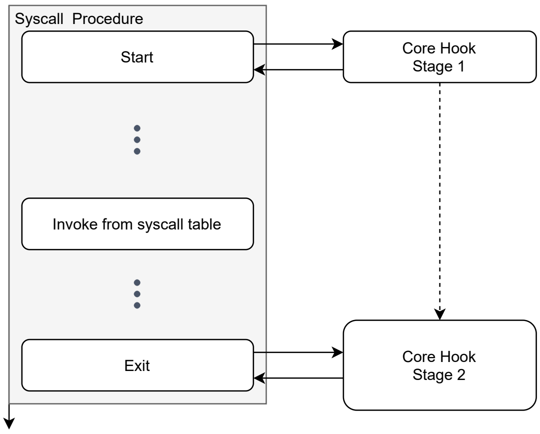

Figure 2: The two tages of *core hook*

the system calls processed in a relatively long period. So, there will be multiple system calls going to different stages of *core hook*.

This problem is solved by the observation that syscall will block the thread in user space. Therefore, we can find a one-to-one mapping from thread number to a system call event at any moment. I maintain a table to save these correspondences in second stage, and also get its first parameters from stage 1.

3.4 Filter

The `filter` part is a relatively simple component that requires information about the caller of the syscall from the Linux kernel. Then it will perform filter by pre-passed conditions. Last, it passes this filtered information on to the next part.

3.5 Record Buffer

The *record buffer* is intended to act as a transit between kernel space and user space. Therefore it has two parts located in kernel space and user space respectively. One of the simplest designs is to maintain a daemon in user space constantly querying and retrieving the data stored in the buffer, and then dumping the data to a file. However, we note that this introduces a huge amount of overhead, mainly due to frequent file io. Placing a larger buffer in user space would also solve this problem, but ${PROJECT_NAME} do not want to introduce a large impact on the entire system.

Therefore, *record buffer* is designed to keep fetching the buffer occupancy, and then dumps the whole buffer only after finding that the buffer occupancy has reached a threshold.

4. Implementation

4.1 Core Hook

The part of *core hook* aims to hook system call, i.e., inject custom code at the begin and the end of a system call. By leveraging *core hook*, It is easy to get the return value of the system call and the changes to the parameters. Linux has provided many different approaches to achieve it, such as *ptrace*, *auditd*, *Kprobe* and *tracepoint*.

4.2 Filter

4.3 Shared Memory

5. Related Work

There is a large amount of related work dedicated to cpaturing syscalls. In this section, I discuss some representative examples and describe how ${PROJECT_NAME} differs.

- **Pinplay** is a ...
- **REPT** ...
- **rr** ...
- **DTrace** ...
- **sysdig** ...

Bibliography

[1] MICHAEL H, DENYS V. Getrandom(2) —Linux manual page[EB/OL]. 2021 [2021-03-27]. https://man7.org/linux/man-pages/man2/ptrace.2.html.

[2] O' CALLAHAN R, JONES C, FROYD N, et al. Engineering Record and Replay for Deployability[C]//2017 USENIX Annual Technical Conference (USENIX ATC 17). Santa Clara, CA: USENIX Association, 2017: 377-389.

[3] CHEN Y, ZHANG S, GUO Q, et al. Deterministic Replay: A Survey[J/OL]. ACM Comput. Surv., 2015, 48(2). https://doi.org/10.1145/2790077. DOI: 10.1145/2790077.

[4] ALTEKAR G, STOICA I. ODR: output-deterministic replay for multicore debugging[C/OL]//Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles - SOSP '09. Big Sky, Montana, USA: ACM Press, 2009: 193 [2021-03-21]. http://portal.acm.org/citation.cfm?doid=1629575.1629594. DOI: 10.1145/1629575.1629594.

[5] BHANSALI S, CHEN W K, de JONG S, et al. Framework for instruction-level tracing and analysis of program executions[C]//VEE '06: Proceedings of the 2nd international conference on Virtual execution environments. New York, NY, USA: Association for Computing Machinery, 2006: 154-163.

[6] DUNLAP G W, KING S T, CINAR S, et al. ReVirt: enabling intrusion analysis through virtual-machine logging and replay[J]. ACM SIGOPS Operating Systems Review, 2003, 36(SI): 211-224.

[7] DUNLAP G W, LUCCHETTI D G, FETTERMAN M A, et al. Execution Replay of Multiprocessor Virtual Machines[C/OL]//VEE '08: Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. Seattle, WA, USA: Association for Computing Machinery, 2008: 121-130. https://doi.org/10.1145/1346256.1346273. DOI: 10.1145/1346256.1346273.

[8] CUI W, GE X, KASIKCI B, et al. REPT: Reverse Debugging of Failures in Deployed Software[C]//OSDI'18: Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation. Carlsbad, CA, USA: USENIX Association, 2018: 17-32.

[9] MONTESINOS P, HICKS M, KING S T, et al. Capo: A Software-Hardware Interface for Practical Deterministic Multiprocessor Replay[J/OL]. SIGARCH Comput. Archit. News, 2009, 37(1): 73-84. https://doi.org/10.1145/2528521.1508254. DOI: 10.1145/2528521.1508254.

[10] TORVALDS L. Linux (5.11)[EB/OL]. 2021 [2021-03-27]. https://github.com/torvalds/linux/releases/tag/v5.11.

[11] MATHIEU D. Using the Linux Kernel Tracepoints —The Linux Kernel documentation[EB/OL]. [2021-03-30]. https://www.kernel.org/doc/html/latest/trace/tracepoints.html.

[12] BOVET D P, CESATI M. Understanding the Linux Kernel[M]. 3rd ed. [S.l.]: OReilly, 2006.

[13] CORBET J, RUBINI A, KROAH-HARTMAN G, et al. Linux device drivers[M]. 3rd ed. [S.l.]: O'Reilly, 2005.

[14] University of California. Syscall(2) - Linux manual page[EB/OL]. 2021 [2021-03-28]. https://man7.org/linux/man-pages/man2/syscall.2.html.