User-level Deterministic Replay via Accurate Non-deterministic Event Capture

Jeongtaek Lim¹, Hosang Yoon¹, Hyunmin Yoon², Yoomee Ko¹, and Minsoo Ryu¹
¹Department of Computer Science and Engineering, Hanyang University, Seoul, Korea
²Department of Electronics and Computer Engineering, Hanyang University, Seoul, Korea
{jtlim, hsyoon, hmyoon}@rtcc.hanyang.ac.kr, cut20032002@naver.com, msryu@hanyang.ac.kr

Abstract – Recently, software that directly affects human life has increased and developed. For example, we use software on almost everything, such as cars, home appliances, and smart phones. Since bugs can have a devastating effect on people, ensuring software stability is becoming important. Many debuggers have been developed to help developers eliminate bugs. However, the debuggers did not help much in eliminating bugs such as synchronization faults. These bugs are caused by non-deterministic events. Non-deterministic events occur at unpredictable timing and affect the behavior of user processes. In this paper, we approach captures and replays non-deterministic events of single thread to make the thread run deterministically and will be useful for software debugging. We must replay the events at the same time as we capture all the non-deterministic events. For this, we get the help of the ARM debug architecture without having to modify the user program.

Keywords: deterministic replay, non-deterministic events, system call replay, thread communication replay

1 Introduction

Recently, software has been applied to various technologies. Software is becoming larger and more complex to provide more functionality. As a result, software bugs are more likely to occur. Research at Cambridge University says that the economic damage caused by the bugs around the world is about $312 billion per year. Especially, more than a million lines of automotive and aircraft software are directly related to the user’s life. Accordingly ensuring software stability is becoming very important. However, it is difficult to guarantee the software reliability with the conventional debugging and testing methods.

Software bugs affect user programs in a variety of ways, and to varying degrees. Bugs can have a minor impact on user programs, but they can also cause serious errors, such as the termination of a program. Also some bugs that occur or do not occur depending on the progress of the program. So it is almost impossible to find and remove all bugs.

Several debuggers such as GNU debugger (GDB) have been developed to help developers remove these bugs. GDB provides many features that allow developers to track and modify the process to eliminate bugs. However, these debuggers are not very helpful in resolving synchronization fault. These bugs are caused not only by problems inside the user thread but also by external factors. In this paper, we call these factors non-deterministic factors/events. These non-deterministic events occur in a variety of ways, such as an interrupt generated by an external device or communication with an external thread. In order to eliminate bugs that are difficult to detect with debuggers, we must be able to capture, replay, and control non-deterministic events.

Our approach is to record all non-deterministic events that can affect the target thread and to perform a deterministic replay of the target thread based on the recorded events. To accomplish this, we need to analyze what non-deterministic events are in the target thread. The thread uses the system call to request external hardware resources or to request kernel services. If we can capture and replay the non-deterministic factors of the system call, we will be able to deterministic replay many external events. Also, we need to find non-deterministic factors that do not use system calls. Threads can access the same data by sharing memory and can send signals to each other to communicate. Non-deterministic events of these specific cases are also captured and replayed to achieve our goal.

For our approach, we get help from the ARM debug architecture. However, there is no need to modify the user program for our approach.

2 Overview of Record-Replay

2.1 Main idea

The goal of our record-replay approach is to log all non-deterministic factors that can act on the target user thread during the record, and then reproduce the event at the exact point in the replay. Non-deterministic factors affecting target thread are unpredictable events and data originating from outside the program. These factors are either caused by
system calls or by communication with processes and threads. So our paper suggests a record-replay method of non-deterministic factors of system calls and communication with processes and threads that affect target thread.

2.2 Recording mechanism

2.2.1 System call record

All system calls do not cause non-deterministic factors. Therefore, there is no need to record-replay all system calls. For efficient record-replay, it is necessary to classify system calls that have non-deterministic factors. First, process control functions such as fork() and clone() are deterministic in terms of the thread since they access only the values of the process control block. Second, memory control functions such as mmap(), brk(), and munmap() are functions that manage virtual memory. These functions are deterministic because they are not affected by hardware or external events. The previous two kinds of system calls do not need to be recorded. However, I/O functions such as read() and recvmsg() are used when interacting with peripheral devices or interacting with other processes. These functions can be considered non-deterministic because it can have an unexpected effect on the target thread. So our system records I/O functions.

We do not need to record the entire function process. Our system only records factors that may have a non-deterministic effect on target thread. I/O system calls write data to the user memory area and the return value of the system call affects the thread. For example, the read() system call writes the data read from the kernel to the user space and returns the data size. Through recording of these data, non-deterministic factors of the system call can be recorded.

2.2.2 Communication event record

In fact, inter-process communication using message queue or using socket can be handled by system call record because it operates using system calls. However, Inter-process communication using signal does not use system call. And, after organizing shared memory, the process does not use system call to access shared memory. So in this section we will show you how to record inter-process communication using signal and inter-thread communication using shared memory.

First we need to know how to handle the signal. Linux does not deal with a signal immediately when it receives a signal. It saves the received signal in the signal queue, enters the kernel mode for various reasons such as system call and interrupt, and handle the signals stored in the queue one by one before returning to user mode. Accordingly, in order to reproduce the signal, it is important to know not only which signal, but also when it was handled. When the signal is handled, our system will record the signal number and the point in time at which it is handled. This point of time information includes the user mode register of the target thread and the instruction counter of the target thread when the signal is handled. In inter-thread communication using shared memory, the recording method is different. Because it shares the memory address space, communication is possible without the help of the kernel. This makes it difficult to detect all accesses to shared memory. The page table needs to be modified to detect access to the shared memory area. We change the access permission bit of the page table in the shared memory area. Now, access to all shared memory causes the exception. We can get the data of the shared memory to be accessed in the exception handler.

2.3 Replaying mechanism

2.3.1 System call replay

I/O function system calls can be emulated only by copying data. In order to efficiently replay, we do not actually perform the system call but replay only by copying the data in the user memory area and copying the return value. It skips the in-kernel operation of the system call and can be much faster than the record. Process control functions and memory control functions do not emulate because their in-kernel process is important.

2.3.2 Communication event replay

Inter-process communication using system call can be reproduced by system call replay. First, we will explain the replay mechanism of signals, not the Inter-process communication method using system calls. The signal log contains the number of the signal and the time the signal was processed. We get the help of the ARM debug architecture to reproduce exactly when the signal is processed. Set a breakpoint at the instruction address in the user mode register of the log. An exception occurs when the program counter reaches the instruction address where the breakpoint is set. Then if match in the Exception handler as compared to the log of the instruction counter to the current instruction counter, sends a signal to the target thread. These operations work the same as the signal processed in the record and the replay is complete.

Inter-thread communication using shared memory is similar to the record mechanism. Accessing the shared memory area raises an exception and replaces shared memory data with data in the log. If process access memory after this operation, you access the same data as the record time. In this way, a replay of communication is performed.

3 Implementation

3.1 Non-deterministic event record system
We first illustrate our overall record implementation in Figure 1. As shown in Figure 1, Target thread invokes a system call to use the service provided by kernel. If hardware resources are needed during the system call processing, the kernel requests the hardware. Target thread also communicate with other threads. This communication affects the behavior of the target thread. The Record-Replay module in the kernel catches only external factors that have a non-deterministic effect on the target thread and creates event list in the order processed. The list are then sent to the Record-Replay monitor process. The Record-Replay Monitor process stores this list as the log in the storage.

![Figure 1. Overall implementation of record](image)

Our approach is supported by the ARM debug Architecture. Breakpoint debug event based on instruction address match, instruction address mismatch, or context match. Figure 2 shows when a breakpoint event occurs. And, the instruction counter indicates the number of instructions executed by the target thread. We can see exactly how much the target thread has been executed.

When the target thread invokes a system call, The Record-Replay module creates a SYS log. This log contains only the system call number. SYS log is necessary to process the signal replay at the correct time during replay to eliminate the error of the breakpoint technique. Kernel uses the copy_to_user function to write to the user memory area during system call processing. The Record-Replay module creates a CTU log by capturing the data from the kernel and the user memory address to which the data will be written in the copy-to-user function. Finally, the system call is finished and the Record-Replay module captures the return value of the system call and stores it as a RET log. Thus, we can record all non-deterministic factors of the system call.

After finishing in-kernel work, the kernel handle the pended signals just before returning to user mode in kernel mode. When the kernel gets pended signals, the Record-Replay module captures the signal number, instruction counter, and user mode register and stores it in the SIG log. The user mode register is the user process context that is stored before the jump to kernel mode.

When the target thread configures the shared memory, the Record-Replay module changes the access permission bit of the shared memory page to No Access. Then, when the target thread accesses the shared memory, data exception occurs. This allows Record-Replay module to catch all accesses to shared memory. When an exception occurs due to shared memory access, the Record-Replay module restores the permission bit so that it can access shared memory. Next, capture the data from the shared memory that the target thread was trying to access. Finally, set a breakpoint mismatch event in the user mode instruction address where the exception occurred. When context return to user mode in kernel mode, the target thread can access shared memory by executing the instruction that caused the exception. After that, the program counter moves to the next instruction address, a breakpoint mismatch event occurs. The Record-Replay module then changes the permission bit of the shared memory page table to No Access in order to redetect access to the shared memory.

![Figure 2. Breakpoint event occurrence](image)

### 3.2 Deterministic replay system

Figure 3 shows our overall replay implementation. The Record-Replay monitor process takes the saved logs and sends them to the Record-Replay module. Logs are loaded in exactly the order in which the events occurred. During replay process, The Record-Replay module controls all non-deterministic external factors. Obtaining hardware resources and in-kernel work may be omitted.

When the system call is invoked, remove SYS log. It then checks if it is a system call that needs to be replayed. If it is not a system call that needs to be replayed, then it is executed normally. However, if the system call should replay, Record-Replay module writes data to the user memory area if CTU log to use in this system call exists, and replaces the system call return value with the RET log. This series of operations completes the emulation of the system call. After processing the system call, check what kind of log is to be processed next. If the next log is not SIG log, it returns to user mode without any further action. If the next log is a SIG log, we must determine if we need to replay the signal before returning to user mode. If the user mode register and the instruction counter in the SIG log match current state, send a
signal to the current process. If it does not match, use a breakpoint to find the match state. If a breakpoint is set in the user mode instruction address in the log and then the instruction counter is also matched in the breakpoint match event handler, The Record-Replay module determines that the current state is the same as when the signal was recorded. It then sends the signal. If the system call and signal are processed in the same user context, signal processing should be done after the system call processing. Breakpoint events are processed prior to system call processing. Since signal processing can take precedence over system call processing, check whether the next log is a SIG log or a SYS log. If the next log is a SYS log, then the breakpoint exception handler does nothing so the signal can be processed after system call processing.

Inter-thread communication replay with shared memory is similar to the way to record. At the time of record, Record-Replay module capture the data to be accessed. At the time of replay, Record-Replay module change the data to be accessed into the data in the log.

4 Conclusions

In this paper, we implement a way to catch all non-deterministic factor in a single thread and replay efficiently. Moreover, our implementation does not need to modify the target user program. We can also reproduce the software bug generated by non-deterministic factors, analyze the log to find out what the problem is. Using our implementation with a debugger such as the GNU debugger will make it easier to analyze bugs caused by non-deterministic factors. On the other hand, the debug features supported by each architecture differs, so it is architecture dependent. However, our approach is applicable if any architecture can catch and replay with non-deterministic factors. In addition, if we can replay the scheduling events on our system, we can record/replay all threads in a multi-threaded environment. If we can capture and replay more events we need, our approach can scale to various environments.

5 Acknowledgement

This research was supported by the MSIP(Ministry of Science, ICT and Future Planning), Korea, under the ITRC(Information Technology Research Center) support program(IITP-2017-2012-0-00628) supervised by the IITP(Institute for Information & communications Technology Promotion) and partly supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIP) (R0114-16-0046, Software Black Box for Highly Dependable Computing) and partly supported by the MISP(Ministry of Science, ICT & Future Planning), Korea, under the National Program for Excellence in SW(R7116161027000102) supervised by the IITP(Institute for Information & communications Technology Promotion)

6 References

