Reproducing Non-deterministic Bugs with Lightweight Recording in Production Environments

Nan Wang1, 2, Jizhong Han1, Haiping Fu1, 2, Xubin He3, Jinyun Fang1
1. Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
2. Graduate University, Chinese Academy of Sciences, Beijing, China
3. Virginia Commonwealth University, Richmond, Virginia, U.S.A.
{wangnan06, hjz}@ict.ac.cn, haipingf@gmail.com, xhe@vcu.edu, fangjy@ict.ac.cn

Abstract

Reproducing non-deterministic bugs is challenging. Recording program execution in production environments and reproducing bugs is an effective way to re-enable cyclic debugging. Unfortunately, most current record-replay approaches introduce large perturbations to either environments and/or execution flow, in addition to performance penalty and high storage overhead, which make them impracticable to be deployed in production environments.

This paper presents Snitchaser – a fully user-space record-replay tool which can faithfully reproduce bugs by replaying system calls which are recorded with negligible perturbation and recording overhead. This is achieved by 1) a novel, lightweight system call interception mechanism without patching the binary instructions to reduce the perturbation to execution flow; 2) system call latch to save signal semantic; 3) periodic checkpointing to reduce the storage overhead.

Snitchaser focuses on bugs caused by asynchronous events on heavily loaded, high throughput servers. Experimental results show that Snitchaser is capable of reproducing non-deterministic bugs efficiently at nearly no performance penalty. We also present two case studies on dealing with existing bugs in Lighttpd – a popular software used in many large scale systems.

1 Introduction

Programs often fail due to software bugs. When bugs arise, programmers need to identify and fix them promptly. Consequently, debugging skill is considered to be one of the most important programming skills. Our cyclic debugging skill helps us to isolate and identify bugs by reproducing them repeatedly and examining the bug conditions. During this debugging cycle, we may employ many different debugging tools. However, reliably reproducing the bugs is the base of other debugging actions. We are well trained to deal with deterministic bugs, and are familiar with some particular types of bugs such as memory corruption. However, with the rapid emergence of distributed systems, parallel computing and high throughput requirements, today we have to fight those bugs triggered by some uncontrollable conditions, such as scheduling, ordering, network thrashing and non-blocking I/O. Those bugs arise and disappear randomly, which makes them very hard to be isolated. They are known as heisenbugs [13] to indicate their non-determinism. Recent work [24] shows that when dealing with concurrency bugs (most of them are heisenbugs), time to reproduce a bug dominates the entire debugging process.

Traditionally, to capture and isolate these non-deterministic bugs, developers need to deduce the root cause by reviewing logs and analyzing source code manually. It is labor-intensive and time-consuming, thus inefficient. Moreover, fine-grained log overwhelms human’s ability, and coarse-grained log is unable to provide enough information.

Record-replay [17, 32] is proposed to help developers to reliably reproduce bugs. By recording interactions between the process and its environment, and later replaying all non-deterministic operations, developers can faithfully track down the errors. Record-replay actually re-executes the target program, so all program run-time states can be reconstructed and examined.

Record-replay converts heisenbugs to traditional bohrbugs [13] and enables developers to use their cyclic debugging skill. With record-replay, developers can use their familiar debugging tools, such as GDB [3], to examine the changing of variables as well as the control flow, which makes developers deal with bugs more easily.

However, although developers can reliably reproduce some bugs using record-replay, those bugs initially emerging at production systems are still needed to be reproduced at least once in development systems without record information, which is challenging. This is because the root cause of most non-deterministic bugs are heavily correlated to the scale, pressure and complexity of the whole system, so it is hard to emulate those bugs by traditional pressure testing in development systems. We call this issue a first reproducing problem.

To avoid the first reproducing problem, recording in pro-
duction environments is a straightforward idea. We have
the following design guidelines for a successful tool to reproduce bugs in production environments.

- **Small perturbation:** The recorder should introduce as small perturbation as possible. Some perturbations, such as special hardware and patched kernel, modify execution environment seriously. Such modification introduces extra overhead and complexity, as well as potential security hole, which is intolerable to production systems. Some perturbations, such as binary instrumentation and semantic modifications of signal processing, change the behavior of the program, require extra attention of users, and may cover some bugs up.

- **Efficiency:** The recording phase should not cause too much performance penalty. Users of large, critical production systems will not allow any add-ons to be deployed on their systems which may heavily reduce system performance.

- **Low space overhead:** The recorder should not generate too large logs to overwhelm production system’s storage. Never shrinking logs will finally take up all usable storage space in a 7x24 production system.

- **Transparency:** The recorder should not rely on special version of executable binary. Users tend to choose “release version” systems, so employing debug version binaries is not a good commercial policy. Moreover, to build a replayable binary always needs modification of source code, which is not always an easy work, especially for legacy code.

To the best of our knowledge, few of existing approaches meets the above 4 guidelines.

In this paper, we propose Snitchaser, a lightweight record-replay tool which enables developers to record in real production environments. By using the following techniques, Snitchaser achieves small perturbation, low performance penalty, low space overhead and transparency to new and legacy programs:

- It develops a novel, fully user-space system call interceptor, which can intercept and log system calls without any modification or patch to execution binary instructions. This makes the target instruction sequence identical to the untraced execution. The performance penalty of Snitchaser’s system call interceptor is negligible for heavily loaded network programs;

- Different from previous user space approaches, Snitchaser pays attention to some special system calls and signal handling to save their semantics. *System call latch* is adopted to allow signals to arise anywhere without obvious overhead;

- Its space overhead is clamped to an configurable size via periodic checkpoint. Experimental results show that the overhead of checkpointing is small.

Snitchaser is suitable for modern high throughput programs such as Lighttpd [20] and nginx [38] which are deployed in many large scaled systems. In these programs, bugs are always caused by race conditions of nondeterministic system calls and signal handling.

The rest of this paper continues as follows: Section 2 discusses related work. In section 3 and 4 we give the implementation of Snitchaser in details. We analyze its performance in Section 5. We conclude with our final comments and present our future work in section6.

2 Related work

Many record-replay mechanisms have been proposed since 1990s [10], including content- and ordering-based approaches [32]. In content-based record-replay, all incoming nondeterministic events and their corresponding data are recorded during execution, and are replayed for debugging. In contrast, ordering-based approaches require a group of programs to record only the outgoing of nondeterministic events, and regenerate the inter-processes messages during replay.

Ordering-based policy greatly reduces the log size and the record overhead. Some implementations apply ordering-based solutions[21, 39]. However, ordering-based policy has serious disadvantage: it requires whole system replay because all data need to be regenerated. Therefore, for large cluster, ordering-based replay is too expensive to be practical. MPIWiz [43] developed a mixed solution: ordering-based policy for inner-group events, and content-based policy for inter-groups messages. Nowadays, content-based approaches are dominant.

This paper focuses on content-based approaches. According to the interception level, they can be categorized into the following groups: architecture level approaches, system call level approaches and API level approaches. Architecture level approaches focus on low level nondeterminism, such as hardware interruptions and sequence of memory accessing. System call level approaches focus on processes, recording each non-deterministic system call. API level approaches record non-determinism at API level, suitable for some specific programs. Overall speaking, the higher the interception level is, the smaller performance penalty it introduces, but the less accuracy it can achieve.

2.1 Architecture level approaches

Most architecture level approaches are complex and expensive. For example, FDR [42] focuses on debugging whole system. It architecturally supports replaying the last
second of full system execution before the crash. It intercepts interruptions and memory accesses, employs 1.3 MB of on-chip hardware, and generates 34MB logs in 1 second, as well as some GBs main memory image. BugNet [29] is another architectural approach. Recently some architectural approaches don’t employ special hardware: ReVirt [11] and [19] are based on virtual machines; [9] introduces binary translation. Despite some successes, those software solutions are not suitable for production environments due to their poor performance. For example, [9] slows down the program execution by about 5.66 to 17 times. This is unacceptable in production systems.

Although Snitchaser isn’t an architectural approach, it inherits some of their ideas. Like FDR [42] and BugNet [29], Snitchaser also uses periodic checkpointing to support continuous recording.

### 2.2 System call level approaches

System call level approaches such as Flashback [37] and Jockey [35] intercept and log non-deterministic system calls for replay. From their perspectives, nearly all non-determinism comes from system calls, so intercepting at system call level is enough. Compared with architecture approaches, system call level approaches can achieve lower performance penalty and less perturbation. However, for the non-determinism which comes from other sources, system call approaches have to do extra work to deal with them. For example, neither of the above two approaches allows signals to arise outside system calls.

Snitchaser is a system call level approach, it inherits fork-based checkpointing technique from Jockey and Flashback.

### 2.3 API based approaches

In some situations, all non-determinism comes from specific libraries. For example, MPI programs communicate with each other through MPICH2 library; Google applications share a small corpus of ubiquitous threading, control flow, and RPC library [36]; nearly all Linux programs use glibc. For those programs, intercepting at API level is good for record-replay. Liblog [12], [14, 33] WiDS [23] and R2 [15] are those approaches. API based approaches usually generate smaller logs and can achieve higher performance than other level approaches. However, there are still some non-deterministic sources other than libraries which can make these record-replay approaches fail. Additionally, some APIs are much more complex than system calls, which makes API level record-replay hard to be implemented. For example, Linux has no more than 350 system calls, but glibc contains more than 1500 exported API functions. Some approaches such as WiDS [23] and R2 [15] require developers to modify their source code to adopt the new interface, which is intolerable in practical situations.

### 2.4 Multithreading and high throughput programs

Currently research community focuses on multi-threaded programs on multi-core platforms. For example, PRES [30], ConCrash [25], Capo [28] and ConMem [44] are those researches. Most of them rely on recording low level events such as data race, thread interleaving and lock ordering. As we discussed above, low level recording always introduces heavy performance penalty. Some approaches such as PRES [30] allow developers to record at higher interception levels to reduce overhead, and reproduce the bug scenario by thoroughly checking interleaving space. Some researches provide theories to reduce interleaving space [44].

On the other hand, under the pressure of high throughput requirements, multithreading model has shown its vulnerability [22, 40, 18]. Many applications return to single threading model, such as Lighttpd [20], Squid [41] and nginx [38]. Although memcached [1] still use pthread, it doesn’t do IPC via shared memory. The key techniques of these high throughput programs are event-driven model, non-blocking I/O and asynchronous I/O. These techniques amplify race condition problems caused by non-deterministic system calls and signal handling. Unfortunately, few of existing research discusses these problems.

<table>
<thead>
<tr>
<th>Name</th>
<th>Perturbation</th>
<th>Performance</th>
<th>Storage</th>
<th>Transparency</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Architecture approaches</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FDR [42]</td>
<td>*</td>
<td>0.01%</td>
<td>34MB/s</td>
<td></td>
</tr>
<tr>
<td>BugNet [29]</td>
<td>*</td>
<td>&gt;1.5x</td>
<td>&gt;1GB/s</td>
<td></td>
</tr>
<tr>
<td>[9]</td>
<td>1</td>
<td>5.66 – 17%</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>System call approaches</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flashback [17]</td>
<td>*</td>
<td>18%</td>
<td>60MB/h</td>
<td></td>
</tr>
<tr>
<td>Jockey [35]</td>
<td>o</td>
<td>4.50%</td>
<td>40B/call</td>
<td></td>
</tr>
<tr>
<td><strong>API approaches</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>liblog [12]</td>
<td>o</td>
<td>1.5x</td>
<td>10.6%</td>
<td></td>
</tr>
<tr>
<td>WiDS [23]</td>
<td>o</td>
<td>18%</td>
<td>60MB/h</td>
<td></td>
</tr>
<tr>
<td>R2 [15]</td>
<td>1</td>
<td>1.60MB/b</td>
<td>Ne2</td>
<td>Ne3</td>
</tr>
<tr>
<td><strong>Configuration</strong></td>
<td>PRES [30]</td>
<td>56.6%</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1 summarizes the approaches we have discussed, and none of them satisfies the requirements for a production environment: low perturbation, low penalty, low storage overhead, and transparency. Motivated by this observation, we propose Snitchaser, an efficient non-deterministic bug reproducing tool for production environments.
3 Architecture and process

In this section, we discuss the design and implementation of Snitchaser in details. First, we describe the architecture and workflow of Snitchaser. We then discuss the implementation in details in subsections 3.2, 3.3 and 3.4. The last subsection discusses the processing of system calls.

3.1 Architecture

Snitchaser consists of three main components: a shared object (wrapper), an execution loader (loader), and a debugging loader (replayer).

Figure 1. Snitchaser’s components and process flow

Figure 1 illustrates the architecture of Snitchaser and the workflow of Snitchaser. During the production execution, user loads the target executable using loader. The loader initiates the process, injects wrapper into the new process’ memory space, then redirects system calls to wrapper’s interceptor. After that, all following system calls are redirected. Each time a system call issued, the interceptor appends an entry into log file. The first checkpoint is generated before real execution. When the log size exceeds a user-defined threshold, new checkpoints are generated and wrapper switches its log file. The old checkpoints and logs can be removed by a daemon or manually.

The replay phase starts from a previous (usually the last) checkpoint before a bug arises. At the beginning of the replay phase, the replayer creates the target process and rebuilds the memory image according to the checkpoint. The process then continues its execution from the position where the checkpoint is built. During the replay, each time the process issues a system call, the call is redirected and the result is loaded from the log, and the original execution will be faithfully recreated, therefore bugs will be reproduced.

3.2 The wrapper

The wrapper is the heart of Snitchaser. During the loading phase, wrapper is injected into target process’ memory image using ptrace() by loader. When the process comes into the main() function, wrapper begins to intercept system calls and never use ptrace() again.

The key technique wrapper used to intercept system calls is the wrapper vDSO. vDSO [34] (Virtual Dynamically-linked Shared Object) is the system call mechanism provided by Linux kernel 2.6. It allows processes to issue system calls by calling an entry in a kernel-provided shared object instead of the traditional interruption gate (int $0x80 in x86). Wrapper redirects all vDSO system call to itself by changing the initial stack and cheating the ld facility.

Figure 2 explains the wrapper mechanism.

The vDSO entry is known by the program through a kernel-provided entry in the initial stack. Wrapper modifies it to its own system call entry before the ld begins to work. The following system calls are then redirected, and wrapper can do its recording work before and after the real system calls are issued.

However, the vDSO facility is not ready before the main() function is called, all system calls issued by ld are through traditional int $0x80 gate. Therefore, Snitchaser divides each process’ execution into loading phase (before main()) and running phase (after main()). System calls in loading phase are intercepted by ptrace.

3.3 Logging and checkpointing

Each time when a process issues a system call, the wrapper appends an entry into the log file. While the program runs, the log size grows. To prevent the growing logs from overwhelming the storage, the wrapper periodically generates checkpoints.

The main content of checkpoint is the memory image of the running process. It also contains some essential information which is needed for replaying, such as the state of registers, the i387 FPU state, the heap and signal handler information. Unlike checkpoints BLCR [16] and starfish [8], Snitchaser doesn’t care about file descriptor tracing, network connection tracing and usage information because such data are accessed via system calls and will be recorded.

To reduce the overhead of checkpointing, Snitchaser uses fork-based checkpoint like libckpt [31], flashback [37] and Jockey [35].

Checkpoints and logs are all regular files and can be copied from production environments to development environments. Therefore, Snitchaser allows developers to re-
liably reproduce production bugs in their development environments.

3.4 Replay

When a program crashes, developers begin to reproduce the bugs by replaying the program from the last checkpoint to the crash point. replayer is responsible to rebuild the process’ memory image according to the checkpoint, then reset the wrapper to replay mode. After all preparations are completed, users can use GDB to attach to the replay process, then debug as usual. During replay, each time when a system call issued, wrapper reads the results from log without actual invocation. This mechanism is transparent to developers.

3.5 System call processing

The basic processing of system calls is to record their return values. It is enough for many system calls such as getpid, access and link because they affect the execution only by their return values. System calls such as read and stat modify a block of memory, Snitchaser saves those data and restores them during replay. A few of system calls needs to be treated specially, including memory mapping calls, process control calls and signal handling calls.

For memory mapping system calls (mmap, mmap2, munmap and mprotect), the replayer must reissues same system calls to guarantee following memory accesses to behave identically to the original execution. mmap and mmap2 extend memory space and may fill them with data from specific file. Snitchaser allow user to save all data into log for accuracy, or only save the path of mapped file for recording performance. We prefer the latter policy because real programs rarely map frequently changing files.

close, fork and execv are process control system calls. Currently Snitchaser only support fork and clone without sharing memory. In parents’ context, Snitchaser treats them as trivial system calls and only records their return values. In children’s context, Snitchaser is able to generate a new checkpoint to keep tracing children by switch to a new log file. Children tracing can also be turned off.

sigaction, rt_sigaction, sigreturn and rt_sigreturn are signal handling system calls, which are discussed in next section below.

4 Signal handling

Signal handling is the most obscure, complex and error-prone mechanism in Unix and Linux programming. It is the source of many non-deterministic bugs. Most of existing system call and API level approaches are unable to process signal handling correctly, or only allow signals to arise at specific points, which changes signal’s semantics. Changing of signal semantics introduces perturbation, which makes program behave different from untraced execution, so may cover some bugs.

Snitchaser treats signals carefully to guarantee signals to behave identically to untraced execution.

The basic technique Snitchaser uses to process signal is to take over signal handling. Snitchaser intercepts signal handling system calls sigaction and rt_sigaction. When target process installs signal handlers, wrapper replaces the handler by Snitchaser’s wrapper signal handler. Therefore, each time when signal arrives, wrapper signal handler has a chance to record signal information into log file before transferring control to the real signal handler. Snitchaser also wraps rt_sigreturn by setting SA_RESTORE, which can notify Snitchaser when user’s signal handler returns. Wrapper signal handlers and wrapper sigreturn handler append some data into log file. We will discuss them in subsection 4.2.

Only recording signal arising and signal handler returning events is not enough for replay. The biggest problem for replayer is to determine when to transfer to signal handlers. Although the signal frame contains the address where the signal arises, this address is insufficient because one instruction can be executed many times in a loop.

Snitchaser creates a new checkpoint, then switches to a new log file when signals break normal instructions. This policy always generates many checkpoints and log files and makes debugging work harder. On the other hand, our experience shows that most of signals terminate system calls and arise right after those system calls returning. Snitchaser can record such signals into log to prevent checkpointing and log switching. To prevent wrapper signal handler from disturbing original log sequence while recording enough information for replayer, Snitchaser employs system call latch mechanism.

4.1 System call latch

We list pseudocode of our system call handler below. This process takes effect each time when a system call is issued.

```
1  /* signals may arise here */
2  base = counter;
3  /* signals may arise here */
4  DISABLE_SIGNAL;
5  counter ++;
6  proc_handler:
7  ENABLE_SIGNAL;
8  /* signals may arise here */
9  issue_realsyscall;
10  /* signals may arise here */
11  DISABLE_SIGNAL;
12  post_handler;
13  counter --;
14  ENABLE_SIGNAL;
15  /* signals may arise here */
```

The two variables of base and counter are employed to judge signal position. Signal handler wrapper identifies whether a signal breaking a system (can record signal into log file) or not (need checkpoint switch) by comparing those two variables. In previous pseudocode, signals arise at line 8 and line 10 are identified as “breaking a system call”.

After a signal arises, wrapper signal handler saves those 2 variables into signal frame (using the retcode field) and rebalance them. When signal handler returns, their values are retrieved from signal frame. The value in base indicates the nest level of signal handlers.

The macros of DISABLE_SIGNAL and ENABLE_SIGNAL create 2 atomicity blocks. All logging operations are done inside them to prevent wrapper signal handler from disturbing log sequence. The pre_handler in the first block appends the system call number into log file, the post_handler in the second block appends the return value and extra data of the the system call into log. post_handler also appends a no-signal-mark into log before effective data, we discuss the mark later.

DISABLE_SIGNAL and ENABLE_SIGNAL issue rt_sigprocmask() to reset signal mask, which makes one system call issue 4 system calls to prevent rare events. Snitchaser provides a signal relax mode, which turns off those 2 macros. Our experience shows that signal relax mode is safe most of the time, and can improve recording performance.

4.2 Replay of signal handling

During replay, when process issues a system call, replayer first checks system call number in log file, then check the no-signal-mark. A no-signal-mark immediately follows system call number means the system call completed normally during execution. A signal-mark at that place means the system call is broken by a signal. In this situation, replayer build the signal frame according to log, then transfer control flow to correct user-defined signal handler. When handler return, replayer restart the processing of the broken system call.

To achieve that, the wrapper signal handler records a signal-mark into log as well as signal frame information. The wrapper sigreturn handler appends the system call number again.

5 Evaluation

We have implemented Snitchaser on Linux in about 16k lines of C code and verified its efficiency. This section evaluates its performance and demonstrates our real debugging experiences using Snitchaser.

5.1 Performance

We test Snitchaser on two computers connected via a 100Mb/s Ethernet switch, both use kernel 2.6.25.15 with a 2.2GHz Intel Core 2 Duo E4500 CPU and 2GB memory. We test LMbench3 [27] and some real programs.

LMbench 3 [27] is a test suit for Unix system performance. We use 2 benchmarks to measure memory accessing and network performance. In these tests, Snitchaser stores its logs and checkpoints into a memory-based file system mpfs. The sizes of LMbench’s checkpoints are usually less than 3MB, and the log sizes are limited to 10MB. The results are listed in table 2. All listed results are the average value of three tests.

The results show that Snitchaser introduces no extra cost to memory access. It introduces less than 10% overhead to TCP data transferring even if we deploy server and client on same machine. The results of lat_tcp also show a trend that the closer the environment to real production environment, the lower performance penalty Snitchaser can achieve: if we deploy server and client on different machines, Snitchaser introduces no performance penalty.

We also evaluate Snitchaser with some real programs: lame [26], wget [2], Lighttpd[20] and php [6]. We test lame by encoding a 7 minutes .wav file to .mp3 file. We test wget by downloading a very large file, and the speed we list here is reported by wget itself. During wget testing, we disable the ht-timing feature in glibc to ensure wget never issue rdtsc instruction. We test Lighttpd by accessing a small, static web page via apache ab. We test php by accessing a small php page (generated by a simple phpinfo() function) by ab. The php test also uses Lighttpd as http server. In those 2 http benchmarks, we set ab’s -c parameter to 20 and -n parameter to 10000. To emulate real production environment, all network tests use two machines. The results in table 3 show that Snitchaser introduces nearly no extra overhead to real production systems.

Table 3 also shows checkpoint’s storage overhead. In the php test, Lighttpd and php generate 3 checkpoints. The size of Lighttpd’s checkpoint is 3.2MB, the total size of two php workers’ checkpoints is 18.1MB. However, php FastCGI sometimes recreates its workers: during 10000 requests, it creates 21 workers. Each worker generates a checkpoint about 10MB (their sizes are different). The value listed in table 3 is the total size of all checkpoints. On the other hand, only two workers are active. All other workers exit normally and leave their checkpoints and log files in storage. In our production environment, we use a shell script to remove those checkpoints.
We also evaluate the performance penalty of checkpointing by setting the log threshold to 128kB. In this configuration, processes checkpoint 80 times more quickly than the default configuration. The results in table 3 show that even we clamp the threshold to such a strict, impractical size, the overhead is still no more than 30%.

5.2 Debugging experiences

In this subsection we share our debugging experiences with Snitchaser. We have used Snitchaser to solve some real bugs, and below we discuss two bugs in Lighttpd.

5.2.1 Lighttpd bug 2052

Lighttpd bug 2052 [4] is related to php and FastCGI. It’s noticed that PHP FastCGI sometimes fails to kill its children [5]. The bug was confirmed at 2007. Developers suspected that the root cause was related to signal handling, and provided some patches. However the bug wasn’t solved until Aug. 2009.

We also suffer from this problem and want to check it in our production system. To adopt Snitchaser, we change the scripts for starting Lighttpd and php-cgi, then let the system execute as usual. The traced executions generate lots of checkpoints and log files. The biggest challenge we meet is to identify the buggy execution from many traces. We achieve that goal by using a daemon to check whether the children fast-cgi processes still alive or not after the corresponding Lighttpd process ends for a while.

After we finally find the buggy trace, debugging becomes very straightforward. We first check and confirm that php-cgi processes have never received SIGTERM, and Lighttpd process has never sent it. By setting breakpoints at signal handlers and replaying under GDB’s control, we find that Lighttpd process is terminated by a SIGTERM but its signal handler (Lighttpd kill php-cgi in its signal handler) is never executed. We also realize that buggy Lighttpd processes are always terminated right after their initiation.

After three minutes hacking we realize the root cause of the bug: Lighttpd initializes php-cgi before it installs signal handlers. If it gets killed during or right after php-cgi initialization, it terminates in the default behavior, never kill its children. Fixing this problem is also not very hard: bringing the installation of signal handlers forward is enough. According to our further communication with Lighttpd community, this problem may not be the only cause of the bug. However it is still a big defect of the software.

This is not a real hard-to-reproduce bug, it happens too often to force end users write scripts to work around. The cause of it is not very complex, neither. We believe the reason why it isn’t fixed for more than 2 years is the difference between production environment and development environment. Development environment is relatively simpler compared to production environment, developer hard to realize that a Lighttpd worker may be killed right after its creation. In contrast, Snitchaser’s bug scenario is generated in production environment.

5.2.2 Lighttpd bug 2217

Lighttpd bug 2217 [7] is related to CGI processing. Someone reported that if CGI processes exit too fast, a few of http requests get timeout. Developers were unable to reproduce the problem and provided a relatively complex patch to “fix” this problem without fingering out the root cause.

We adopt Snitchaser to deal with this bug, and finally find the root cause. The bug is caused by the race condition related to non-blocking I/O and connection state machine.

Lighttpd introduces epoll to process large number of connections, and uses state machine to deal with each file descriptor. To avoid waiting, Lighttpd uses non-blocking I/O when receiving CGI result from pipe. The pipe is also controlled by epoll state machine. When mod_cgi reads data from pipe, read() may returns normally, returns 0 or returns EAGAIN because of non-blocking I/O. Returning of 0 indicates the finish of receiving, mod_cgi marks the connection as “finished”. Returning of EAGAIN causes Lighttpd to wait for incoming data or incoming “HUP” event in epoll state machine. If epoll receives a HUP event from the pipe, Lighttpd also marks the connection “finished”. Lighttpd retrieves CGI children’s states when sending http responses. If a child terminates, it removes the corresponding pipe from epoll state machine.

The race condition arises when all effective data have been read out but the last read() gets an EAGAIN instead of 0. In this situation, the CGI child exits normally because all its data have been sent. Lighttpd detects the death of the child and removes the pipe from epoll state machine when sending http responses. Therefore, the incoming “HUP” event cannot be received, so the corresponding connection cannot be set to “finished”. Such a unfinished connection doesn’t close the TCP socket, which makes the client side timeout.

Snitchaser plays an important role when dealing with this bug. It allows us to replay the execution many times. We first check accept() and close() to identify the connections which cause timeout. We then use the address of corresponding connection structures to trace their processing (we can do that because the memory state is identical between different replays). We also use Snitchaser to trace some normal executions to understand the correct behavior.

6 Conclusions and future work

This paper focuses on one of the most challenging problems in debugging – bug reproduction. In this paper, we
propose Snitchaser, a record-replay tool for production environments to address the first reproducing problem. It can faithfully reproduce the bugs caused by asynchronous events such as signals, non-blocking I/O and message passing. Therefore, it is suitable for current high throughput programs such as Lighttpd and nginx. Moreover, 1) it introduces nearly no perturbation to production systems, which is guaranteed by preventing expensive extra requirements to production environments, avoiding touching the instruction sequences, and faithfully retaining the semantics of signal processing; 2) its light weight system call interceptor intercepts and logs system call with negligible performance penalty. 3) its storage overhead is restricted by periodic checkpointing. Experimental results show that Snitchaser is efficient as evidenced by low overhead even for heavily loaded server programs. Our experiences also show its power on debugging some bugs in popular production programs. We published the source code of Snitchaser, which can be downloaded on http://gitorious.org/snitchaser.

In our future work, we will focus on multi-threading debugging. We plan to use Snitchaser’s log as execution sketch, and introduce heavy weight record-replay mechanisms in replay phase to reveal buggy thread interleaving.

7 Acknowledgement

This work is partially supported by the National Natural Science Foundation of China (Grant No. 61070028 and 61003063) and Knowledge Innovation Project of the Chinese Academy of Sciences (Grant No. KGXC2-YW-131).

References

[1] memcached - a distributed memory object caching system.
[38] Sysow, Igor. nginx.