TORNADO: A Novel Input Replay Tool*

Frank Cornelis Michiel Ronsse Koen De Bosschere
Department of Electronics and Information Systems
Ghent University
Belgium

Abstract *This paper presents TORNADO, a fully operational tool that enables us to replay the non-deterministic input of real world applications. We present a new technique to trace write operations done by the OS kernel to user space. Using this technique we were able to construct a replay tool capable of replaying a large class of applications with an acceptable overhead of less than a factor 2.*

Keywords: input replay, nondeterminism

1 Introduction

Although a number of advanced programming environments, formal methods and design methodologies for developing reliable software are emerging, one notices that the biggest part of the development time is still spent while debugging and testing applications.

One of the reasons is that debugging and testing is still a human activity which is hard to automate. Another reason is that most programmers still stick to arcane debugging techniques such as adding print instructions or watchpoints or using breakpoints. These techniques are completely inadequate in the development of modern computer software that is highly multi-threaded, distributed, and interactive (e.g. a web browser). As this type of software is highly nondeterministic, no two executions can be guaranteed to be the same, and hence traditional cyclic debugging techniques fail. Finding bugs in these circumstances is more a matter of luck, than a matter of skills, and takes too much time.

The nondeterminism in modern software has a number of causes:

- the input used by a program (e.g. data read from disk, a network packet, mouse movement, ...) cannot always be guaranteed to reoccur during a re-execution.
- shared variables used by multi-threaded programs introduce so-called data races.

The data race problem has been studied extensively in the past, and numerous solutions have been proposed, going from synchronization replay [7] to logical thread scheduling replay [3]. This paper will focus solely on the nondeterminism caused by the non-repeatable input consumed by the program. This input is normally provided to an application by the operating system, e.g. an application uses a *system call* to request the OS kernel to perform an I/O operation.

In order to enable programmers to use the classical cyclic debugging techniques for nondeterministic programs, so-called *record/replay* methods and tools have been developed. Such a tool *records* information about the nondeterminism that is encountered during a program execution in a *trace file*. During subsequent replayed executions the information from this file is consulted in order to guarantee a faithful *replay* of the recorded execution. As these re-executions are equivalent to the recorded execution, cyclic debugging is now possible, e.g. we can use intrusive debugging techniques in order to pinpoint the exact location of the bug without having to worry that the bug will disappear all of a sudden due to the probe effect [5].

In order to be useful, such a record/replay tool should exhibit the following properties:

- the overhead of the tool should be low, both in time and space. If the time overhead is too high, bugs can be missed due to the probe effect. If the space overhead is too high, the tool will consume too much I/O bandwidth, reducing the I/O bandwidth available to the application.

*In proceedings of The 2003 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'03). Published by CSREA Press.

- the tool should be easy to use. E.g. a tool that requires recompiling an application is useless if the application uses proprietary libraries that are only available in binary form.

This paper describes a record/replay tool (called TORNADO) that deals with program input (called input replay in the remainder of this paper) that exhibits these two properties. The tool has been developed for the Linux kernel and is composed of two parts: a kernel patch and a user library. The patch has to be installed once and the user library uses the patch to perform input replay. No instrumentation, recompiling or relinking of the user application is required.

The paper proceeds as follows: in Section 2, we discuss some concepts on input. Section 3 describes the design of our TORNADO tool while Section 4 presents benchmark results. Section 5 gives an overview on related work, and we conclude in Section 6.

2 Input concepts

In this section a number of important concepts regarding input replay will be discussed. Some of these concepts and the problems they cause are specific for the operating system for which we developed TORNADO (being Linux, or more generally UNIX), e.g. the concept of user and kernel space.

2.1 Partial Input & Partial Output

When replaying the input of a program we will close the normal input channels and (transparently) force the program to use the input we traced during the record phase. Nevertheless, it can sometimes be useful to make the program receive input from the real environment during the replay phase. E.g. when it is certain that some data input source does not alter over time (e.g. a read-only file) it is safe to receive input from the original data sources which saves us from recording that data. These partial input redirections can also be used for delta debugging. Basically, delta debugging sets up subsets of the original circumstances, and tests these configurations whether the failure still occurs [4]. One of the circumstances which can cause program failure is the program input.

Analogously, when replaying a program one often will appreciate it that the program still produces some kind of output since this is often used as criterion for a correct replay from the user's point of view (black-box effect). On the other hand, some generated output can be undesirable because it might lead to inconsistencies within the system. For example, as output operation a program could insert an entry in a database. When the database is constrained to unique tuples a replaying program instance will make the database to error when actually requesting the insertion.

As it is the case with our replay module, adequate facilities should be present in a record/replay system to support partial input and partial output during the replay phase. This also has an impact on the record phase as different partial I/O policies may require varying data streams to be recorded.

2.2 User vs. Kernel Context & Kernel Object Mapping

When talking about replay it is useful to view the whole picture in terms of context reproducibility. A program has two contexts: a user context and a kernel context. The *user context* comprises the content of the program memory along with the content of the (user) processor registers. The *kernel context* on the other hand can be thought of as the set of in-kernel data structures necessary to make the user program run. E.g., if an application opens a file using the `open()` system call, the application receives a file descriptor (an integer) which constitutes the user context. Upon opening the file, the kernel also adds more detailed information about the file (size, permission, current file pointer, ...) to the application's kernel context.

What we try to procure during a replay is to reproduce the user context for the program. We do not need to reconstruct an identical kernel context when replaying a program. In fact, this is most likely not possible when replaying a program; the kernel data structures are shared between multiple processes which often compete in both kernel level as user level for system resources in a very rude fashion. This is where we stumble on a hitch: we're trying to reproduce the user context but most likely the underlying kernel context differs from the original. Given the fact that the user context usually depends upon the kernel context for certain operations we will have to translate between what the (identical) reproduced user context "thinks" of the kernel context and what the actual kernel presents as the replay kernel context to the replaying process.

As it is the case with our replay module, adequate facilities should be present in a record/replay system to support kernel object mapping. Our replay module supports context mapping of file de-

scriptors. Because of the partial input/output on some files (see section 2.1), the file descriptor maps (allocated by the kernel) differ between the record and the replay phase, necessitating file descriptor mapping. Luckily the memory mapping can be fixed between program instances during replay phase, making memory context mapping superfluous.

2.3 System Call Classification

The nature of a system call is very important when it comes to implementing an input replay system which uses the system call level to provide record and replay functionality. When looking at system calls from the perspective of a record/replay system we can categorize system calls according to either their record behavior or their replay behavior. As for the record behavior they fall into the following categories:

- System calls that only generate a return code (e.g. `getpid()`). Nowadays, almost all system calls return at least a return code as indication of a successful execution. Tracing the return codes should suffice in order to be able to replay these system calls.
- System calls that (along with a return code) change a well known user memory area (e.g. `read()`). The address(es) of the changed memory area(s) can be calculated from the system call parameters. Tracing the return codes along with the memory areas (or only the differences between the original areas and the changed areas) should suffice to replay these system calls.
- System calls that (along with a return code) change a *very hard* to discover memory area. Part of this category are system calls like `ioctl()` which offer an extensible interface from the kernel's point of view and thus are likely to be device dependent. Keeping up with all device drivers in order to chart every memory area addressed by all involved system calls would be virtually impossible.

  When trying to replay this kind of system calls one needs a method to acquire the addresses of the memory areas changed by these class of system calls. The technique used in our replay system to accomplish this will be described in section 3.1.

When looking at the replay behavior we can categorize system calls as follows:

- System calls that we do not need to re-execute during the replay phase. Thus we just content-based replay them by injecting their previously traced results (side effects) into the user program. Doing so we are assured that the replay is identical to the original execution since the program is receiving the same data from the outside world as before.
- Some system calls we still want to re-execute during the replay phase for example because we want them to generate their output again. Most likely we are not interested in the input that these system calls produce to-wards the user program. So we just inject the original traced input data into the program and do not bother about what these system calls actual return. Thus we only re-execute these system calls because *we* like them to get re-executed, not because the system badly needs their re-execution.

  Sometimes we also want to re-execute these systems calls to reduce the amount of trace data needed to obtain a perfect replay (see also section 2.1).
- Some system calls need to be re-executed because of their side effects on the kernel context of the user program. Part of these system calls are memory management calls like `mmap()` and `munmap()`. If we neglect the re-execution of these system calls the subset of the kernel context responsible for running the user program will become irreversible inconsistent with the user context. This can lead to unrecoverable errors during the replay phase.

  This category of system calls can be further divided into:

  - System calls that are expected to produce exactly the *same* return codes as during the record phase (e.g. `mmap()`).
  - System calls that are expected to behave equivalently during both record as replay phase: they should return the same error code, or (in case the system call succeeds) they return an equivalent result (e.g. a file descriptor). These system calls should be covered up by the kernel object mapping subsystem of the replay framework (see section 2.2) in order to be able to present the program with the same user context as before.

3 Design and Implementation on Linux

Adopting all of the previously mentioned concepts had a significant impact on the implementation of our input replay tool. This even required a redesign of the input replay tool into two modules: an extended `ptrace` kernel patch (called `exptrace`) and a user library (called `replay`). The `exptrace` patch provides interposition functionality which makes it possible to trace system calls at a very dense granularity while the `replay` module takes care of the actual record/replay functionality. These two modules have been combined into the TORNADO package.

As part of the design process we chose for a model in which the tracing process and the tracer live in separate memory areas. Doing so minimizes the intrusion that the tracer has on the memory layout of the process it wants to record/replay.

In the following subsections we will handle the kernel patch and the user library in more detail.

3.1 Compile-time Instrumentation of Kernel-user Copy Operations

As mentioned in the previous section, most system calls write to well known memory regions, and it is easy to detect these regions by intercepting the system calls. Things are far more complicated for the system calls for which the used memory addresses are not easy to detect if one simply intercepts the system call. In our input replay framework we developed a technique to trace every store operation performed by the kernel to user memory to fulfill this need.

The Linux kernel forces code to use certain memory access primitives (shown in Table 1) for writing to user memory. Thus instrumenting these memory operations suffices to be able to trace every memory write access from kernel to user memory. We

Linux kernel 2.4.20 (i386)	
Name	**File**
`__put_user_u64`	`uaccess.h`
`__put_user_asm`	`uaccess.h`
`__copy_user`	`uaccess.h`
`__constant_copy_user`	`uaccess.h`
`__do_clear_user`	`usercopy.c`

Table 1: *Kernel-to-User write primitives.*

instrumented these write primitives in such a way that they add an entry to a linked list for each store operation. Each entry contains the start address of the memory area written to and the number of bytes written. This technique is —slightly simplified— depicted in Fig. 3.1. Notice we do not

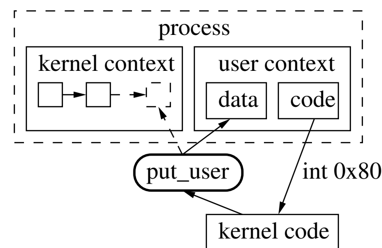

Figure 1: *Instrumentation of write operations.*

store the memory content into these entries because it is always available by consulting the user memory. Another advantage of not storing the data in these entries is that we do not have to take the write history into account when returning the information about the store operations to the tracing process. We can handle the write operation entries as if they occurred simultaneous.

After constructing the linked list of store operations performed by the system call we need to pass this information to the process that requested to trace. Before doing this, we first *normalize* the previously constructed linked list so it contains no redundant information. The normalization process consists of first sorting the original linked list by start address and then removing duplicate memory areas and merging consecutive or overlapping memory areas.

3.2 The record/replay library

We now have two methods to determine the memory addresses used by system calls: either we check the system call parameters, or we use the `exptrace` patch as described above.

The actual input replay system was built as a dynamic link library. Linux can be forced to load this library if an application is started (although the application was not linked against it) using the `LD_PRELOAD` environment variable. Our library contains a so-called `_init` routine that is started before the actual application. This routine will start a new process: the *tracer*. This tracer will attach itself to the original (application) process using the `ptrace()` system call, and hence the tracer will be

woken up for each system call executed by the application.

During the record phase, the tracer checks which system call was executed by the application and then collects the memory addresses used by the system call, either by checking the arguments of the system call, or by using the `ptrace` patch we implemented. For these addresses, the new memory values and the return value of the system call are written to a trace file.

During the replay phase, the memory and return values are simply read from disk and fed to the application, with or without re-executing the actual system call. The replay tool labels a program replay instance as correct deterministic replay by means of the system call pattern.

4 Experimental Results

As mentioned in Section 3, a design was chosen in which the tracer and the process that is getting traced live in separate memory areas. This design choice resulted in extra overhead because of the additional context switches needed to move the trace data between the tracer and the process we trace. Nevertheless, this drawback does not countervail against the necessity of letting the process we record/replay to have its own memory space since every form of intrusion can break the replay.

In order to minimize this additional overhead some classical techniques like I/O trace buffering and caching of user context objects were adopted to increase the performance of the replay module. We also boosted the performance of the main loop of the tracer by extending the Linux kernel with a new experimental system call: `ptrace_wait`. This system call provides the basic mechanism to build a tracing tool and resulted in a $\pm 49\%$ faster minimal tracer main loop.

When it came to benchmarking the input replay tool some problems were encountered as it is very hard to find suitable benchmarks that are providing realistic input requests. Most benchmark suites do not focus on realistic I/O patterns. On the other hand, real world (interactive) applications that exhibit realistic I/O behavior require input replay themselves in order to be suitable as benchmarks because of their nondeterministic nature. This "chicken and egg" problem has been solved by using regular benchmark suites focusing on:

- I/O performance of OS primitives (micro-benchmarks),
- execution time of non-interactive standard applications.

The results of the micro-benchmarks will be discussed first, followed by an overview of the benchmarks with some real world non-interactive applications.

To measure the I/O performance of OS primitives we did use (some of) the HBench-OS benchmark tests [2]. Although no changes to the source code of the HBench benchmark tests were needed, adequate record/replay environments had to be setup for each of the tests to be able to compare non-traced with traced benchmark executions. The results of these benchmark tests have been summarized in Table 2. In this table the slowdown between a non-traced and a traced execution is shown. The "Trace size/iteration" column denotes the average bytes of data needed to allow for a deterministic replay. When studying these numbers one notices that the replay tool provokes the worst slowdowns on the low-level OS primitives (the `lat_syscall` micro-benchmarks). This is mainly caused by the costly context switches that are needed between the process and the tracer to replay a single system call. Of course no real world application exhibits such a weird behavior as these micro-benchmarks do (our worst case scenario) and thus regular applications are not subject to the same slowdowns as shown later on. This overhead could be diminished by handling the entire replay from within the OS kernel but this would also greatly reduce the ease of developing a generic replay tool. Also responsible for the overhead is the partial I/O functionality (see section 2.1) and the kernel object mapping (see section 2.2). The slowdown is of course also proportional to the trace data bandwidth required for a deterministic replay. This is especially important for the `mmap()` latency test (see `lat_mmap`, Table 2). When re-executing the `mmap()` system call during the replay phase, it is not a necessity to re-inject the memory pages into the process and thus no tracing of these pages during the record phase is required which results is less overhead compared to a normal content-based replay of `mmap()`. Finally one also notices that the benchmark latency tests focusing on entire subsystems like RPC[1] (which are closer to real world application behavior) are reporting a smaller overhead compared to micro-benchmarks who are only testing one single system call at a time. This is because the overhead of the replay functionality is less a decisive factor compared to the overall latency provoked by those sub-

[1]Remote Procedure Calls

Micro-benchmark	Slowdown	Trace size/iteration (bytes)
`lat_syscall write`	8.36	6.08
`lat_syscall getpid`	8.60	5.03
`lat_pipe`	3.00	12.16
`lat_mmap`	7.41	1048.58
`lat_mmap` (re-execution)	5.00	20.07
`lat_fs create`	1.20	48.13
`lat_proc`	2.29	72.70
`lat_udp`	2.15	42.09
`lat_tcp`	1.97	39.01
`lat_rpc`	1.82	264.19

Table 2: *HBench-OS 1.0 results for tracing operating system primitives.*

systems. This is especially the case for the TCP and UDP latency benchmarks (again see Table 2). Since the TCP network stack is more complex than the UDP network stack the micro-benchmarks report a smaller overhead of the input replay tool on the TCP latency test.

We also benchmarked some standard non-interactive applications which are closer to real world (interactive) applications than the micro-benchmarks are. The results of these tests have been summarized in Table 3. The following applications were used: `latex` which is a typesetting tool, the GNU `gcc` C-compiler, the `mcs` C#-compiler from the Mono project[2], and the file compression tool `gzip`. Further in this table the "Untraced" and "Traced" columns contain the execution times in seconds. When taking a closer look at these numbers one notices that the slowdown most of the time stays below a factor of 2. This is rather good especially when looking at the size of the trace data that is required to allow for a deterministic replay. As it was the case for the micro-benchmarks, re-execution of I/O system calls during the replay phase will also here result in less trace data to be captured to allow for a deterministic replay. This is especially the case for the `ls` directory listing application (see the last rows of Table 3) and also shows up as a general lower slowdown on other applications.

5 Related Work

Until now, input replay tools have been primarily used for testing graphical user interfaces (GUI). There exist a number of commercially available tools (e.g. CAPBAK from Software Research Inc, JavaStar from Sun and Rational Robot from Rational Software) that enable this, but they focus solely on mouse and keyboard input, e.g. they expect files to remain unchanged.

A tool that is much more comparable to our TORNADO tool is jRapture [8]. jRapture allows recording and replaying of all types of input for Java programs. The tool accomplishes this by providing modified versions of the Java API classes that interact directly with the underlying operating system, e.g. the `FileInputStream` class. Of course, if a Java application uses another class file with its own I/O routines implemented using JNI, jRapture will not be able to record/replay this input.

An interesting technique for finding changed memory regions (e.g. after a system call) is using the virtual memory access protection facility of the underlying OS. By providing a suitable segmentation fault handler, such a technique is able to find changed memory regions by comparing memory pages. This technique is used by TreadMarks [1] and *Compressed Differences* [6]. The main difference between this technique and ours is that TORNADO does not need to interfere with the virtual memory subsystem of the OS to find out where the system is writing data to the user space. With our technique we actually just let the system tell us where it did write data.

6 Conclusions & Future Work

With our fully operational TORNADO input replay tool one is able to replay real world applications exhibiting nondeterministic behavior. The replay tool also supports network replay and thus re-

[2] www.go-mono.org

Application	Execution time (s)		Slowdown	Trace size
	Untraced	Traced		(MiB)
`latex paper.tex`	0.23	0.47	2.04	2.77
`gcc libreplay.c`	1.07	1.45	1.36	6.38
`mcs Hello.cs`	0.69	1.13	1.64	8.29
`gzip afile.tar`	1.96	2.77	1.41	15.22
`ls -R /lib`	0.37	0.73	1.97	5.11
`ls -R /lib` (re-execution)	0.37	0.67	1.81	3.36

Table 3: *Benchmark results for tracing some real world applications.*

play of X-Windows GUI applications (e.g. `xedit`). The power of our tool lies within its interposition technique. By using an altered kernel we were able to attach to what is called a *single point-of-access* within the system, being the system call interface. This is of paramount importance for one to be able to record and replay a system.

Despite our effort more work should be done in this area to eliminate the drawbacks present record/replay tools are subject to, being overhead in both space and time. Our tool already has an acceptable time overhead of less than a factor of 2 but the space overhead still needs some attention. Once these problems are solved, it will become possible for input replay systems to be deployed more generally as for example in systems submitting automatically generated bug reports.

A Availability

The tool can be downloaded from `http://www.elis.rug.ac.be/~fcorneli/`.

References

[1] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. In *IEEE Computer*, February 1996.

[2] A. B. Brown and M. I. Seltzer. Operating system benchmarking in the wake of lmbench: A case study of the performance of NetBSD on the Intel x86 architecture. In *Proceedings of the 1997 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, June 1997.

[3] J. Choi and H. Srinivasan. Deterministic replay of java multithreaded applications. In *Proceeding of the SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 48–59, Welches, Oregon, August 1998.

[4] H. Cleve and A. Zeller. Finding failure causes through automated testing. In *Fourth International Workshop on Automated Debugging*, Munich, Germany, August 2000.

[5] J. Gait. A probe effect in concurrent programs. *Software - Practice and Experience*, 16(3):225–233, March 1986.

[6] J. Planck, J. Xu, and R. Netzer. Compressed differences: An algorithm for fast incremental checkpointing. Technical Report CS-95-302, University of Tennessee, August 1995.

[7] M. Ronsse and K. De Bosschere. RecPlay: A fully integrated practical record/replay system. *ACM Transactions on Computer Systems*, 17(2):133–152, May 1999.

[8] J. Steven, P. Chandra, B. Fleck, and A. Podgurski. jRapture: A capture/replay tool for observation-based testing. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 158–167. ACM Press, 2000.