Transparent Mutable Replay for Multicore Debugging and Patch Validation

Nicolas Viennot Siddharth Nair Jason Nieh
Columbia University
nviennot@cs.columbia.edu, ssn2114@columbia.edu, nieh@cs.columbia.edu

Abstract

We present DORA, a mutable record-replay system which allows a recorded execution of an application to be replayed with a modified version of the application. This feature, not available in previous record-replay systems, enables powerful new functionality. In particular, DORA can help reproduce, diagnose, and fix software bugs by replaying a version of a recorded application that is recompiled with debugging information, reconfigured to produce verbose log output, modified to include additional print statements, or patched to fix a bug.

DORA uses lightweight operating system mechanisms to record an application execution by capturing nondeterministic events to a log without imposing unnecessary timing and ordering constraints. It replays the log using a modified version of the application even in the presence of added, deleted, or modified operations that do not match events in the log. DORA searches for a replay that minimizes differences between the log and the replayed execution of the modified program. If there are no modifications, DORA provides deterministic replay of the unmodified program.

We have implemented a Linux prototype which provides transparent mutable replay without recompiling or relinking applications. We show that DORA is useful for reproducing, diagnosing, and fixing software bugs in real-world applications, including Apache and MySQL. Our results show that DORA (1) captures bugs and replays them with applications modified or reconfigured to produce additional debugging output for root cause diagnosis, (2) captures exploits and replays them with patched applications to validate that the patches successfully eliminate vulnerabilities, (3) records production workloads and replays them with patched applications to validate patches with realistic workloads, and (4) maintains low recording overhead on commodity multicore hardware, making it suitable for production systems.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging; D.2.4 [*Software Engineering*]: Software/Program Verification; D.2.7 [*Software Engineering*]: Distribution, Maintenance, and Enhancement; D.4.5 [*Operating Systems*]: Reliability

Keywords Deterministic Replay, Mutable Replay, Multicore, Debugging

ASPLOS'13, March 16–20, 2013, Houston, Texas, USA.

1. Introduction

As applications grow in complexity, software bugs have become increasingly common and more difficult to reproduce, diagnose, and fix. Aggressive release schedules exacerbate the problem, resulting in frail software that requires patches to fix problems that occur in the field. Resolving a bug typically starts with reproducing it in a controlled environment. Because the common approach of conveying a bug report is often inadequate for tricky, nondeterministic bugs, record-replay has been developed to capture application bugs as they occur and deterministically replay the bug at a later time, removing the burden of repeated testing to reproduce the bug.

Despite an abundance of research on using record-replay systems for debugging [2, 3, 7, 11, 18, 19, 21, 22, 26, 27], most works have focused on bug reproducibility and have had limited or no support for diagnosing and fixing bugs. Debugging almost always requires modifying the program, whether by adding print statements, testing if a change fixes the problem, or some other method. However, previous record-replay systems do not allow the recorded execution to be replayed with any modifications to the application, with a couple exceptions. A handful of systems do allow some new code to be run in the middle of a recording, but they do not support changes to the application state [7, 12], which limits the utility of these systems for debugging and validating changes.

To address this problem, we introduce DORA, a mutable record-replay system which allows a recorded execution of an application to be replayed with a modified version of the application. Mutable record-replay provides a number of benefits for reproducing, diagnosing, and fixing software bugs. For instance, mutable record-replay can replay a version of the recorded application that is recompiled with debugging information, reconfigured to produce verbose log output, or modified to include additional code instrumentation such as print statements.

Mutable record-replay can also replay a recorded application execution of a production workload using a patched version of the application. This is useful for both application developers and system administrators. An application developer can use a recording of a bug when developing a fix. Replaying the recording on a modified application speeds up debugging and provides a novel way of validating bug fixes for nondeterminstic bugs, which can otherwise be time consuming and difficult. For example, a developer who writes a patch can test it by taking a recorded execution of the exploit on the original application and replaying it using the patched application to quickly verify that the patch closes the vulnerability instead of painstakingly regenerating the exploit for each attempted fix of the problem.

System administrators often worry that applying patches will break their applications. Mutable replay allows administrators to independently test patches on production workloads. An administrator can record the unpatched application in production, apply the patch to an offline version of the application, and then replay

the recorded execution using the patched application. If the replay succeeds, the administrator will be more confident that the changes will not introduce regressions.

Mutable replay complements traditional quality assurance testing. Quality assurance provides broad coverage but fails to handle many corner cases, which is why bugs arise in production in the first place. In contrast, mutable replay isolates actual bugs that occur in production. These bugs can be timing and configuration dependent, so some surface very rarely. This coverage is often not possible with traditional testing due to nondeterministic program behavior. Furthermore, mutable replay provides fast turnaround time, enabling a bug to be replayed quickly and directly tested against application changes that attempt to fix the problem. This not only speeds up debugging, but also provides a way to validate bug fixes for nondeterminstic bugs, which can otherwise be much more time consuming and difficult.

DORA consists of three components: (1) a recorder that records application execution to a log, (2) a replayer that can replay a modified version of the application using the log, and (3) an explorer that uses the replayer to find the execution of the modified program that best corresponds to the log file. The recorder and replayer build upon our previous deterministic record-replay engine, Scribe [16].

The recorder operates primarily at the interface between applications and the operating system (OS) to transparently record an application's nondeterministic interactions. It avoids imposing unnecessary timing and ordering constraints that would hinder mutable replay with a modified application. To aid mutable replay, the recorder also logs deterministic interactions to detect and resolve any differences between the recorded application execution and the replay of a modified version of the application.

The replayer replays a previously recorded execution using a modified version of the application, matching events from the original log with the actions of the modified program. If the application used for replay is the same as the one recorded, the replayer provides deterministic replay of the unmodified application. However, if the replayed application's behavior diverges from the original's, the replayer gathers information for the explorer about the new code path the program was trying to execute and waits for instructions on how to proceed. Because it operates at the OS level like the recorder, the replayer has access to sufficient OS semantics to understand why a replay diverges from the original execution and can leverage these semantics to help the explorer.

The explorer evaluates several possible execution paths to find a successful mutable replay. It performs a best-first search for an execution of the modified program that is as close to the original execution as possible according to some cost function d. It begins by replaying a recorded execution on a modified program. When the replay diverges from the original execution, the explorer tries to determine why. For example, suppose the modified program made an unexpected `printf()` call. This could be a new call to produce debugging information, or it could simply occur earlier than expected because code was deleted. The explorer chooses the most promising possibility and communicates its decision to the replayer. This process repeats until a successful execution is found.

DORA is designed to handle an wide range of real-world programs, including multi-threaded applications. It can support a broad range of useful application changes, but cannot support arbitrary changes; major changes to the process layout or shared memory layout are not supported. Despite this limitation, DORA is useful in a wide range of real-world use cases for testing, debugging, and validating application changes. In fact, we even found a previously unknown bug in Apache using DORA [1]. DORA's usefulness in practice makes sense given that bug fixes tend to be relatively small and rarely change core application semantics [13, 29].

We have implemented a DORA Linux prototype that runs on commodity multicore hardware without changing, relinking, or recompiling applications or libraries. Our experimental results with over thirty different application changes show that DORA can (1) record unmodified real-world multi-threaded applications with less than 10% overhead on multicore hardware, (2) replay applications that have been reconfigured to produce verbose debugging output or modified with added debugging instrumentation, (3) replay real exploits on patched applications to verify that the patches close these vulnerabilities, and (4) replay benchmark workloads to validate application patches and version upgrades despite changes in thousands of lines of code.

We present the design, implementation, and evaluation of the DORA mutable replay system. Section 2 provides a definition of mutable replay. Section 3 describes the DORA recorder. Section 4 describes the DORA replayer. Section 5 describes the DORA explorer and presents an example illustrating the use of the system. Section 6 presents some of the guarantees that can be made about the system's behavior. Section 7 discusses limitations of the current system. Section 8 presents experimental results. Section 9 discusses related work. Finally, we present some concluding remarks and directions for future work.

2. Mutable Replay Concept

Since mutable replay is a previously undefined concept, we begin by presenting a definition. Let e be the recorded execution of some program P. Let E' be the set of possible executions of P', a modified version of P. A mutable replay of e on P' will then be an execution e' in E' such that the differences between e and e' are a result of the differences between P and P'.

The difference between two programs includes not only differences in their executables, but can also include changes in input and environment, such as environment variables, the file system, and host-related information. Note that there is not always a clear mapping from input in the original program to input in the modified program. For example, the original program could read more bytes from `stdin` than the modified program.

Since some executions in E' are intuitively preferable to others, we introduce the concept of a *d-optimal mutable replay*, an execution in E' that is optimal according to a cost function d. The cost function measures the difference between the original execution and the mutable replay. The value returned by d reflects the minimal cost of transforming the execution e_1 into a candidate execution e_2. A lower score is better, scores can be negative, and the score must be the lowest when e_1 is identical to e_2. A d-optimal mutable replay e_d of an execution e on P' satisfies $d(e, e_d) = min_{e' \in E'} d(e, e')$. There is always at least one d-optimal mutable replay for a given execution e and a program P'.

Finding the d-optimal mutable replay is undecidable in the general case. To show this, we first observe that finding the d-optimal mutable replay requires running P' because predicting the executions of a program is undecidable. Suppose P' has an added infinite loop at its beginning. Then, when running P', the replayer will loop infinitely since detecting an infinite loop is undecidable. Thus, finding the d-optimal replay is undecidable.

Even in the subset of cases in which finding a d-optimal mutable replay is decidable, it is still NP-hard with respect to the number of events in the log. Consider a program P' which only adds a `read()` system call of `n` bytes. There are $O(2^n)$ possible results of this call. In addition, arbitrary signals could be delivered between any two instructions. If the program is threaded, they are many possible thread interleavings. Since differences in signal delivery and thread interleaving could theoretically cause radically different behavior, and since determining the future execution of a program is undecidable, a mutable replayer must consider every possibility,

```
int main() {
  printf("%d\n", time(NULL));
  return 0;
}
```

Figure 1. Original program

```
int main() {
  FILE *out = fopen("output", "w");
  fprintf(out, "%d\n", time(NULL));
  return 0;
}
```

Figure 2. Modified program

which is infeasible. Thus, no mutable replay system can efficiently find a d-optimal mutable replay in all cases.

Fortunately, however, many useful changes to programs are modest in size and scope. In particular, bug fixes tend to be relatively small and rarely change core application semantics [13, 29]. The same is typically true of code instrumentation added to a program for debugging. Based on this observation, we designed DORA with a d function that has useful properties for testing and debugging. In this context, Section 8 shows that DORA is able to find d-optimal mutable replays in practice using real-world applications. Furthermore, Section 6 presents guarantees that can be made about the optimality of DORA's approximation algorithm.

A simple example may help further clarify the concept of mutable replay. Figure 1 shows a program that prints the current time in seconds to `stdout`. Figure 2 shows a modified version of the original program that instead writes the output to a file. Intuitively, we want the `gettimeofday()` call in the replay of the modified program to return the same time returned in the recorded execution of the original program. We will show that DORA does this, producing a d-optimal replay with the cost function described in Section 5.

3. Recorder

DORA's recorder builds upon our Scribe work on lightweight OS-level deterministic replay on multiprocessors [16]. The Scribe engine provides four key benefits for mutable replay. First, operating at the OS level avoids tracking low-level hardware nondeterminism that is unnecessary for application replay and would significantly complicate mutable replay. Second, DORA records the execution of system calls in a manner that enables system calls and their effects on the kernel to be fully executed during replay. As discussed in Section 4, this is essential for mutable replay because there are times when DORA must transition processes from controlled replay to live execution to enable mutable replay. Third, DORA's recorder can record the execution of multiple processes and threads with low overhead on production systems. Finally, DORA's recording is transparent to applications. It does not require changing, relinking, or recompiling applications or libraries, and it supports programs written in any programming language.

The recorder operates on a group of tasks (threads and processes), which we refer to as a *session*. DORA records interactions between the session and its external environment, such as incoming network packets and nondeterministic interactions between tasks, in a manner which accommodates application changes during replay. DORA also records deterministic information to help detect changes in an application's execution path during replay.

The recorder saves the recorded execution to a log file. Figure 3 shows the tail of the log file generated by running the simple program in Figure 1. We excluded 83 events related to program initialization, including `execve()` and C library bootstrapping events.

```
// 83 initialization events
--- cut ---
munmap(0xb76e1000, 968e) = 0
rdtsc = 000057ed322904cf
time(NULL) = 0x4f9bd2e7
fstat(1, 0xbff8d684) = 0
mmap(0, 1000, 3, 34, -1, 0) = 0xb76ea000
write(1, 0xb76ea000, 11) = 11
exit(0) = 0
```

Figure 3. Recorded log file of original execution

The last two initialization events are shown. The `rdtsc` event corresponds to seeding a random generator from the C library by reading the time stamp counter of the CPU. The log also includes several system calls and information about their arguments.

3.1 Nondeterministic Interactions

To replay an execution deterministically, the timing and ordering of nondeterministic interactions between tasks must be recorded. To enable mutable replay, DORA must also be resilient to changes in the application behavior which add or remove nondeterministic system calls. We highlight how key interactions involving system calls, signals, and shared memory are handled.

System calls. The ordering of system calls that access shared resources causes nondeterminism if at least one call modifies the resource. For instance, a `write()` and a `read()` on the same pipe are *related system calls* since they access a shared resource and their relative ordering matters. Preserving the order of related system calls during replay ensures that recorded racy behavior is maintained in the replay. To provide this deterministic behavior, DORA operates at the OS level to capture concurrency at the same level of granularity as the OS. DORA leverages kernel code that already serializes access to shared kernel objects to record and enforce a partial ordering of all related system calls. This is in contrast to previous approaches [11, 26] which impose a total ordering of system calls. Such approaches do not scale well for multicore and do not work for mutable replay. Application changes are likely to change the number and order of system calls, making a total ordering of system calls too restrictive for replaying an execution using a modified program.

Shared objects tracked by DORA include inodes, files, filetables, memory maps, process credentials, process states, and system-wide properties such as the hostname and mount points. To keep track of access ordering, every resource is assigned a globally unique identifier and a serial number indicating the order of access. For each resource a system call accesses, DORA increments its serial number and records its identifier and serial number. This allow the replayer to deterministically reproduce the access sequence.

Signals. The delivery time of signals is another source of nondeterminism. Signals are normally delivered in two steps in the kernel. First, the sender process sends a signal to the target process, which is marked as having a signal pending. Second, the target process detects and handles the pending signal when it returns to user space from kernel space. If the target process is executing in user space on another core, an inter-processor interrupt will force it into kernel space, where it will detect the pending signal. Replaying this behavior can be challenging because it requires interrupting the target process at the exact same instruction as during its original execution. Previous approaches [5, 6, 9, 10] have relied on hardware providing a cycle accurate instruction counter [25], but this does not work for CPUs without such counters and does not work for mutable replay. Application changes are extremely likely

to change the number and ordering of instructions, rendering the counter values useless.

To address this problem, DORA defers signal delivery to locations called *sync points*. Sync points are well-defined locations in an execution which deterministically cause the process to enter kernel space, such as system calls, traps due to division by zero, and page faults due to shared memory accesses, as discussed further below. By only delivering signals at sync points, DORA effectively converts asynchronous events into synchronous ones and does not require special hardware or application modifications. This also makes it much easier to replay using an application with modified signals. Note that this behavior complies with the POSIX semantics of signals.

Delaying signal delivery to sync points may introduce some latency in the application. However, our previous work on Scribe [16] shows that sync points occur quite frequently in common desktop and server applications. Our evaluation showed that deferred signals were delayed by less than 100 μs on average with a maximum delay of less than 1 ms, so this latency is generally imperceptible. Furthermore, the vast majority of signals were delivered immediately because the target process was already in a sync point when the signal was sent. This is not surprising given that processes often stay in a sync point for a prolonged period of time. For example, a process blocking on I/O in a system call is in a sync point. While a process is in a sync point, signals are delivered immediately and need not be deferred.

Shared memory. Shared memory accesses cause nondeterminism arising from the order in which threads and processes read and write to the same memory locations. Memory is shared either implicitly, as with threads that share an entire address space, or explicitly, as with processes that share a common memory mapping. Since processes typically access multiple locations on a page during a given time interval because of spatial and temporal locality, DORA tracks memory accesses at a page granularity by managing page ownership with the help of the hardware page protection mechanism. Each shared memory page is assigned an owner process or thread for some time interval. The owner can exclusively modify that page during that interval and treat it like private memory. Thus, DORA does not need to track every memory access during such periods. Transitioning page ownership from one process or thread to another is done using a concurrent read, exclusive write (CREW) protocol on memory pages with some optimizations [16].

To ensure that ownership transitions occur at precisely the same location in the execution during both record and replay, previous approaches [5, 6, 9, 10] have relied on cycle-accurate hardware instruction counters. An ordering this precise is not resilient to application changes, so it does not work for mutable replay. To address this problem, DORA defers ownership transitions until the owner reaches a sync point. As with system calls accessing shared resources, a memory event is logged for the thread that accesses shared memory. The event includes a page identifier and a per page serial number indicating the order of access; this process is similar to the one for system resources.

DORA implements this CREW protocol by creating a *shadow page table* for each thread, which is a private version of the shared page table. If a thread does not have ownership of a page, its shadow page table entry (PTE) access privilege bits are cleared. When a thread tries to access a page owned by another thread, it triggers a page fault, notifies the owner, and blocks until access is granted. Note that while the thread is blocked, it is in a sync point, so it can immediately release pages to other requesting threads, which prevents deadlock [16]. Once the page owner reaches a sync point in its execution, it transfers the page ownership to the thread requesting access, which then returns from its page fault handler. Recording performance can be negatively impacted for applications that exhibit an enormous number of ownership transitions. Nevertheless, DORA performs well for a wide range of real-world applications as shown in Section 8.

3.2 Additional Information for Mutable Replay

In traditional replay, all deterministic information need not be recorded because it will be regenerated during replay, but in mutable replay, the modified application may behave differently even in deterministic sections of code. DORA's recorder stores additional deterministic information in the log to help the replayer detect differences between the original and replayed executions as early as possible. Since DORA provides deterministic replay for unmodified applications, it does not need to record the additional deterministic information in production, but instead records such information afterwards by replaying the original execution and recording additional deterministic information as needed.

As part of this process, DORA records all system calls executed, not just those involved in nondeterministic interactions. DORA records the system call number, arguments, and return value for each system call. For arguments that are pointers, DORA follows the pointer chain to record the actual memory contents, which are used during replay to see if two system calls are equivalent. By recording memory contents instead of the pointer values, DORA is more tolerant of memory layout changes caused by application modifications.

DORA also records the virtual addresses of shared memory accesses, allowing the replayer to match shared memory access to detect divergence. However, this mechanism means that changes to the memory layout of writable shared memory affecting page boundaries can cause DORA to incorrectly replay data races. Fortunately, a recent study of common security patches indicates that a vast majority of application patches [13] do not make such changes.

Finally, when performing mutable replay, the modified application may introduce system calls that are nondeterministic and environment dependent. DORA addresses this issue by recording two additional types of information during the original recorded execution. First, DORA stores additional information for mutable replay to ensure that nondeterministic actions that occur during the modified application replay but not during the original recorded execution are consistent with the recorded execution. For example, DORA periodically records timing information to ensure that time-related functions are consistent. If a new `gettimeofday()` call is present in a replayed execution, the replayer can report a time that is consistent with other values replayed from the recorded execution. Second, DORA records other information about the execution environment so that new system calls in the modified application behave as they would have in the environment in which the program was recorded. For example, the recorder stores host information in case the modified application requests it with a new `uname()` or `gethostname()` call.

4. Replayer

DORA's replayer replays the originally recorded execution using either the original program or a modified program. It requires that the execution is replayed on a machine which supports all the recorded instructions. For example, a program that uses SSE instructions when it is recorded cannot be replayed on a machine without SSE instructions unless it is recompiled to use a different ISA. The replayer uses the recorded log file to generate a separate log file per task. Each task is replayed independently, but the replayer enforces the recorded order of access to shared resources.

A key aspect of the replayer is that it can transition a task or a group of tasks from controlled replay to normal execution. This feature is essential for mutable replay because a modified program may have new code to execute that is not part of the recorded

execution. DORA can run such code at any time because it fully executes system calls and their effects on the kernel during replay. Many other replay systems only emulate the effects they have on userspace [11, 22], but this would prevent normal execution of the application from being enabled in the middle of replay.

As a task executes kernel code, the replayer compares the execution with what is expected in the log file. When the execution *matches* expected events in the log, the replayer ensures it behaves as it did in the original execution. System calls match if they have the same system call numbers and arguments. When an argument is a pointer to a buffer, DORA compares the contents of the buffers instead of the pointer addresses. Shared memory access events match if the access types and page addresses are the same. The replay ends successfully if all tasks terminate after consuming every recorded event. A replay that uses an unmodified application will always end successfully in this manner.

However, if a replaying task is about to execute a code path that does not correspond to the expected event, the replay has *diverged* from the log. The replayer conveys this to the explorer, which determines how the replayer should resolve the divergence. If the explorer determines that the replayer should continue replaying the current log, the unexpected event can be treated in number of different ways to try to resolve the divergence so that later events will match events in the log. For simplicity, DORA treats a divergence as one of two possible types of *mutations*, an *addition* or a *deletion*.

4.1 Additions

An addition is an event added to the program. For example, a program could be modified by adding code that includes a new system call. A new event is most often a system call, but it can also be a new signal or shared memory access. When executing an event not in the log file, DORA has three main responsibilities. First, it must decide when to execute the new event relative to events already in the log file. System calls, signals, and shared memory events are often racy with respect to other processes or threads, so there can be many possible orderings. Second, it must ensure that the semantics of the event are consistent with the semantics of the recorded execution. Finally, it is useful to be able to deterministically reproduce these decisions in subsequent replays, as explained in Section 5.

To handle these responsibilities, the replayer switches the process that encountered the addition from controlled replay into *direct mode*. Direct mode switches the respective process to normal execution and enables DORA's recorder to record the execution. This adds a new event to the log and orders it with respect to other events in the log. The resulting log can then be later deterministically replayed with the same modified program. Once the process completes the additional operation, it returns to regular replay mode; no other process has left regular replay mode. We discuss how this is done in further detail for new system calls, signals, and shared memory accesses.

System calls. When encountering a new system call, DORA executes the call and records the execution as discussed in Section 3. This is possible because the replayer fully executes system calls and their effects instead of just emulating them, ensuring that it is possible to switch a process to normal execution at any time. DORA further instruments various system calls to ensure that the new system call behaves consistently with the recorded execution. The specifics of this depend upon the semantics of the added system call. We highlight system calls that deal with three important types of issues: environmental or timing information, resource allocation, and sockets.

For system calls that request environmental information or timing information, DORA ensures that the return values are made consistent with the information in the original log. This includes `gettimeofday()` and `gethostname()`. For example, if a new `gethostname()` call is executed, DORA already recorded such environmental information and ensures that the name reported is the same as what was already recorded.

For system calls that request new resources, DORA ensures that assigned resources do not conflict with those used by the replayed execution. For example, if a new page in memory is allocated, DORA ensures that its address will not conflict with those used in the original program. If the system call manipulates an existing shared resource, the respective resource serial numbers are renumbered to account for the new event.

DORA simply executes new socket-related system calls during replay except when dealing with data streams originating from outside the session. To deal with data streams, such as external sockets, DORA registers fake backends to the corresponding file descriptor. Socket system calls related to those data streams will simply manipulate the recorded network data. For example, if a `read()` on a network socket is changed to a `recvmsg()` on the same socket, DORA provides the appropriate data. If a new `read()` from a socket tries to access more data than recorded, 0 is returned to indicate end of file.

Signals. When encountering a new signal, such as from a modified application with a new `kill()` call, DORA needs to determine when to deliver the signal to the target process. The replayer does this much as the recorder does. Like the recorder, the replayer defers signal delivery until the target process encounters a sync point. This ensures that signal delivery can be replayed deterministically during subsequent replays.

Shared memory. When encountering a new shared memory access, DORA needs to determine how to interleave the access with other accesses. As in recording, a page fault occurs when a replayed process tries to access a shared page that it does not own, and the process must acquire ownership of the page. Once the process obtains ownership and completes the memory access, it releases ownership to the previous owner at the next sync point to ensure that the access order in the original execution is respected. The new memory events are added to the log file and the original serial numbers are reordered as necessary to ensure that subsequent replays based on this log deterministically perform the memory access in the same way.

4.2 Deletions

A deletion corresponds to the removal of events from the original log file and implies that the unexpected event matches a later event in the log. The replayer deletes the intermediate events. There can be several possible matches for an event if the event occurs several times later in the original log. DORA identifies possible matches and reports each possible match to the explorer. Because it is expensive to process many events and it becomes increasingly unlikely to find a match that will result in a successful mutable replay if an extremely large number of events need to be deleted, DORA imposes a cap on the number of events it can remove for a deletion. The cap is 10,000 events in our implementation. We present further detail regarding deletions that involve system calls, signals, and shared memory accesses.

System calls. Most system calls do not have any side effects in the log file, so not executing a call itself is all that is necessary to delete it. If the system call involves a shared resource, DORA also renumbers the serial numbers for the resource so that its serial number sequence does not contain any gaps. Deleting system calls related to external sockets does not remove the incoming data, since it is preserved as part of the stream of data associated with the resource. Data that is not consumed from a deleted socket system

call will eventually be consumed by other remaining or new system calls.

When a deleted system call was originally associated with the delivery of asynchronous events, the events must be relocated to other sync points. For example, consider a program that has a `SIGALRM` delivery scheduled on a `getpid()` sync point. If the new execution no longer calls `getpid()`, DORA must choose a new sync point at which to deliver the `SIGALRM` signal. DORA postpones the delivery of asynchronous events until the next sync point the application encounters. Choosing the next sync point is better than choosing the previous one because releasing page ownership prematurely would introduce spurious page faults in the application and could prevent DORA from respecting the original page access order. For example, suppose a thread writes to a page and then releases ownership of the page at the following sync point, which is on a system call. If the system call is removed and DORA moved the ownership release event to a previous sync point, it would occur before the access to the page. This access would then trigger a page fault and generate a new ownership acquisition event that may not respect the original ordering. Moving the release of ownership to the following sync point avoids this issue.

Signals. Deleting a signal involves deleting its source, which is typically a system call. System calls that deliver signals require additional consideration because their effects create multiple events in the log file. For example, removing a `kill()` system call must also remove the delivery of the corresponding signal. During recording, DORA associates delivered signals with their sources by using an incrementing global token used across the entire session. When a signal is about to be delivered, DORA waits for the source to be triggered or deleted, which respectively delivers the signal or omits the signal from being delivered.

Shared Memory. When deleting a shared memory access, the corresponding ownership acquisition event should not be executed. However, the previous owner still releases its page ownership so that its behavior is consistent with the original recorded execution. Asynchronous events associated with a deleted shared memory event are relocated to other sync points in the same manner as they are for system calls.

4.3 Going Live

In rare cases, the entire log file is consumed before the modified application terminates. This can occur when the original application crashes, but the patched application avoids crashing. The replayer allows the session to *go live* and entirely transition from controlled replay to live execution. This enables the user to validate the correctness of the patch. Since DORA faithfully replays kernel actions, the system is always in a state that allows it to transition to live execution. Linux namespaces [4] create a consistent environment for processes before and after they go live. Examples of this are demonstrated in Section 8.

5. Explorer

The explorer uses the replayer to search for a d-optimal replay. When the replay diverges, the explorer must determine how to proceed. If this was the first divergence, the explorer decides whether the replayer should consider the mutation as an addition or a deletion. If there were previous divergences, the explorer might also tell the replayer to reconsider a previously detected divergence and explore a different path. In this case, the explorer provides the replayer with a new log file. Thus, the replayer needs no knowledge of the exploration algorithm.

The explorer treats the problem of finding the best mutable replay as a search through a tree T of possible candidate executions from a start node e, the execution of the original program, to one of many goal nodes, which represent executions in E'. This problem is different from most other search problems because (1) expanding a node can result in an infinite loop, (2) there can be virtually infinite goal nodes, and (3) the paths to the goal nodes are not known beforehand because determining the possible executions of a program in advance is undecidable.

Since an exact search is undecidable in the general case and NP-hard even when it is decidable, DORA performs an inexact search using a modified uniform-cost search. For simplicity, DORA only considers additions and deletions. It does not consider, for example, input fuzzing or trying all possible racy paths. The algorithm performs the following steps:

1. Initialize T to contain the root node e.
2. Pick an unexplored execution in the tree with the lowest cost according to the cost function d.
3. Attempt to replay this execution on P'.
   (a) If this replay succeeds, this execution is selected and the exploration concludes.
   (b) Otherwise, the replay diverges on an unexpected event, and new nodes are added to the graph. One node represents an addition and the others correspond to each possible deletion. Each node has an associated log file so that nondeterminism in added system calls is reproduced exactly across replays. Go to step 2.

A useful feature of the explorer is that the end result of a replay up to a given node is recorded. Since this recording can be deterministically replayed, a mutable replay is easily reproducible. Furthermore, it is easy to compare two logs to see the differences between two executions. For example, a developer can compare the log of the originally recorded execution with the log of a mutable replay to understand how application modifications affected the replay.

For simplicity, we have implemented the explorer algorithm by replaying a new execution from the beginning of the log upon divergence. In reality, there is no inherent reason for executions to be replayed from the beginning since each child node's log only differs from its parent node's log after the point of divergence. For example, a checkpoint could be taken just before divergence occurs. Nodes created because of this divergence could replay from the checkpoint instead of from the beginning of execution [14, 15, 20].

While any function satisfying the properties specified in Section 2 can be used for d, we present a simple function that has useful properties for debugging purposes. Since matches are desirable and additions and deletions are undesirable, each match has a cost of $-M$ and each addition or deletion has a cost of $+1$, where $M > 1$. We use a value of 3 for M in our prototype, but the process of selecting a mutable replay was relatively insensitive to the specific value. Using a negative cost for matches means that the explorer is unlikely to backtrack after making many contiguous matches. This also means that the uniform-cost search is not guaranteed to find the optimal replay even amongst the nodes it considers (additions and deletions). We made this decision because the number of nodes it would need to consider to guarantee correctness is exponential. Since the future execution of a program is unknown, even potential executions which seem very unpromising could theoretically match many later events in the log file and obtain a very good score. Thus, the search would effectively become a breadth-first search if d could only return non-negative numbers. Event logs may have billions of events long, so this would not be feasible.

```
  munmap(0xb76e1000, 968e) = 0
  rdtsc = 000057ed322904cf
+ brk(NULL) = 0x9870000
+ brk(0x9891000) = 0x9891000
+ open("output", 577, 438) = 500
  time(NULL) = 0x4f9bd2e7
+ fstat(500, 0xbff8d674) = 0
- fstat(1, 0xbff8d684) = 0
  mmap(0, 1000, 3, 34, -1, 0) = 0xb76ea000
+ write(500, 0xb76ea000, 11) = 11
- write(1, 0xb76ea000, 11) = 11
  exit(0) = 0
```

Figure 4. Mutable replay of modified program

Example. To make this process more clear, we return to the example introduced in Section 2, in which a program that prints the time to `stdout` is modified to write the time to a file. Figure 4 illustrates how the explorer replays the modified program in Figure 2 using the recorded execution shown in Figure 3 of the original program in Figure 1. The explorer starts by replaying the original log file on the modified program. The complete original log file contains 90 events, but Figure 4 omits initialization events, such as `execve()` and bootstrapping code from the C library.

The replayer matches the first 85 events successfully, which are all system call events, resulting in a cost of -255. At this point, the replayer encounters a `brk()` that does not match the `time()` call in the original log file and diverges. Upon divergence, DORA can treat `brk()` as an added system call or search the original log for a `brk()` call and delete intermediate events. Since no other call to `brk()` occurs in the log file, only the addition path is considered, resulting in a cost of -254 and an additional node in the tree of candidate executions.

Following this algorithm, replayer adds two more system calls: another `brk()` and `open()`, resulting in a cost of -252. A `time()` call is executed and successfully matches the expected call in the original log, ensuring that the time returned in the replayed execution is the same as the time in the original log. The match lowers the cost of the execution path to -255.

The changed program then executes an `fstat(500, ...)` in the replaying execution. Although the system call number is the same as the `fstat(1, ...)` in the log, the file descriptors passed as the first argument are different. This is treated as a mismatch. No subsequent matching `fstat()` calls are in the log, so this is treated as an another addition, increasing the cost of the execution path to -254.

Next, the application diverges on the call to `mmap()` since it does not match `fstat(1, ...)`. For the first time in this example, the divergence can lead to a deletion or addition since there is a matching `mmap()` in the original log. The explorer creates two nodes, and explores the unexplored node with the lowest cost. In this case, these two nodes are the only unexplored nodes. The addition node costs -253 since there is a one point addition penalty. The deletion node costs -256 because there is a one point deletion penalty for removing one node and a three point bonus for matching `mmap()`. Therefore, the deletion node is selected. The addition and deletion of `fstat()` is effectively a *replacement* of the system call.

The modified program then runs a `write(500, ...)` in the replaying execution which is different from the `write(1, ...)` in the log. Although the function calls are the same, the file descriptors are different, so this is treated as a mismatch. Since no subsequent matching `write()` calls are in the file, this must be treated as an addition, increasing the cost of the execution path to -255.

Finally, the program calls `exit()`, which diverges from the `write(1, ...)` in the original log. The divergence can lead to an addition of `exit_group()` or a deletion of `write(1, ...)`, matching the `exit_group()` in the original log. The addition node costs -254, the deletion node costs -257, and the unexplored node which added `mmap()` costs -253. The deletion node is selected.

Since the end of the log file has been reached, the explorer has found a successful replay with a cost of -257 and terminates. In this case, the explorer has successfully found a d-optimal replay. Although the explorer cannot prove this, the optimality of this replay is evident given the nature of the application modification.

From this simple example, we can observe that small code changes may significantly impact the behavior interactions of the application with the kernel API. The `fopen()` library call internally calls `malloc()`, resulting in two new invocations to `brk()`. Thus, even minor changes to high-level source code can result in relatively large changes to the low-level executable code.

6. Properties

We can make several useful guarantees about the behavior of DORA for certain classes of application changes.

PROPERTY 1. DORA *deterministically replays the original execution if the program is unmodified.*

In other words, DORA performs traditional deterministic record-replay when the program is unchanged. This property also implies that DORA provides d-optimal mutable replay for all d for unmodified programs.

PROPERTY 2. *If all explored mutations are* safe*,* DORA *deterministically replays all events in the original execution with the modified program.*

For our d, a safe mutation is an addition that does not change any state which is read by the original execution. These additions may store state which is later read, but the original execution must not access this new data. This guarantee is quite useful for debugging because it implies that all behavior in the original program, including race conditions, will be preserved deterministically.

For example, a `printf()` changes the internal state of the program by modifying an internal buffer, but returns the program to its original state, assuming proper newlines. Therefore, adding a `printf()` to debug a race condition always preserves recorded races because it will not change the relative ordering of events in the log. DORA can also handle the creation of new files. Any new calls to `open()` will receive a file descriptor not used by the original execution, so DORA guarantees that the original behavior of the program will be unaffected.

As another example, consider memory changes, for both shared and private regions of memory. First, reading the value of a variable in memory is safe. This is true even if the read is from shared memory and triggers a page fault leading to a temporary ownership transition. Additionally, the program can allocate and write to pages that are unused by the original execution. To avoid conflicts, DORA always assigns new memory allocations to a reserved area that is isolated from the original memory mappings.

PROPERTY 3. DORA *does not guarantee deterministic replay when given a modified application with arbitrary modifications.*

As explained in Section 4, DORA does not evaluate all possible interleaving of additions. For example, when a `printf()` is added in two different threads without locking, the order in which the calls are executed is variable. DORA picks the first possibility it encounters during replay and enforces this ordering for subsequent replays. However, this ordering is not enforced across separate invocations of the explorer.

PROPERTY 4. DORA *can deterministically replay a mutable replay of a modified application.*

DORA's explorer outputs the replay it selects to a log file. Thus, DORA can deterministically replay the original execution of the explorer using the modified program. This enables exact reproduction of a previously found replay, allowing DORA to be used iteratively.

7. Limitations

DORA has several limitations as it has no knowledge of application semantics. As a result, it only supports replay across application changes that do not alter core application or execution semantics.

For example, an exploit might add a new entry to a MySQL database. This entry would be assigned a particular id by MySQL, and would affect the ids of all later entries. A patch that removes the exploit would also remove the created entry, resulting in a mismatch between the ids assigned during replay and the recorded assignments. DORA would not find a d-optimal mutable replay because the core semantics of the execution have changed.

Additionally, DORA currently does not effectively support major changes in the layout of shared memory. If objects are relocated from a page to another, DORA cannot always preserve the original access ordering since DORA manages shared memory at the page level. DORA could be modified to track objects instead of pages by instrumenting the application. Even without this functionality, however, DORA is able to handle some changes to MySQL, which heavily uses shared memory.

Finally, DORA currently does not support process/thread layout changes. For example, if an application was originally recorded with 10 threads running, and is now reconfigured to run with 5 threads as part of the application modification, DORA does not provide a way to find a good mutable replay. Similarly, DORA has difficulty with applications that use green threads as small code changes may result in radically different schedules.

Thus, there are some types of changes for which DORA will not find the d-optimal replay. Fortunately, DORA produces enough information for the user to identify when these conditions occur. This allows the user to distinguish behavior caused by an application change from behavior due to DORA's limitations and makes DORA a useful tool for debugging and validation.

These conditions may seem restrictive, but they are often not an issue in practice because patches rarely change core application semantics. In a study of 60 patches each for MySQL, Apache, OpenSSL, and Squid, 83% resulted in only minor changes to the application behavior. Analysis of various security patches showed that over 75% only changed applications in minor ways [13]. This makes sense, as patches often attempt to fix an edge case in an application.

8. Evaluation

We have implemented a prototype of DORA in Linux. The recorder and replayer run in kernel space while the explorer runs in user space. Although the prototype only instruments a subset of the Linux kernel API, we demonstrated the functionality of this prototype in diagnosing and fixing bugs and measured its performance overhead with nine widely used real-world, multi-process, and multi-threaded applications and 32 different application changes involving thousands of lines of code. For our experiments, we used version 2.6.35 of the Linux kernel, Python 2.6.6, Cython 0.14, and UnionFs-Fuse 0.23. Measurements were done on a set of HP DL360 G3 servers, each with dual 3.06 GHz Intel Xeon CPUs, 4 GB RAM, and dual 18 GB local disks.

We recorded a wide range of applications with various workloads that exhibited bugs, as listed in the third column of Table 2.

Name	apache-upgrade	redis-upgrade
Start Version	Apache 2.2.19	Redis 2.4.1
Upgrades	3 (2.2.20 - 2.2.22)	12 (2.4.2 - 2.4.13)
Commits	277	137
LOC	5179+, 388-	2942+, 1154-
Workload	`httperf 0.8`	`redis-benchmark`

Table 1. Application upgrades

We verified that DORA can deterministically replay the original recorded applications and then replayed the executions with modified applications. Section 8.1 shows that DORA can replay these workloads using reconfigured or modified applications with additional debugging or other instrumentation. Section 8.2 shows how DORA can replay the exploits in Table 2 using patched versions of the applications to help developers verify that the bug patches successfully resolve the problems. It also shows how DORA can replay the workloads listed in the last column of Table 2 using the patched versions of the applications to help system administrators verify that the patches do not introduce errors in production workloads. Section 8.3 shows that DORA can verify production workloads on a series of release upgrades for the applications listed in Table 1. Finally, Section 8.4 presents record-replay overhead for the production workloads.

8.1 Debugging and Diagnosis Techniques

We used a wide range of debugging and diagnosis techniques with the workloads listed in the third column of Table 2. Since our work focuses on diagnosing and fixing bugs, most of the problems involve known security vulnerabilities, as indicated by their Common Vulnerabilities and Exposures (CVE) identifiers. However, the apache-log scenario shows a previously unknown bug in Apache that we found with DORA, and the Redis scenario shows how to add retroactive logging without discussing a specific bug.

For each of these exploits, we show how DORA can be used to diagnose the cause of a bug. We consider debugging techniques an experienced developer might apply to identify the root cause of each problem. Table 3 lists the application changes needed to use various debugging and diagnosis techniques for each scenario. DORA successfully found the d-optimal replay for all of these application changes. Table 3 shows the needed replay mutations.

apache-log was originally intended to show how DORA could be used for retroactive logging, but ended up showing DORA finding a previously unknown Apache bug [1]. We wanted to use DORA to add user agent and referrer information to Apache log files, since these could provide useful usage statistics for a website administrator. Although the default Apache logging configuration will not log this information, DORA records all HTTP header information that the server receives. Thus, an administrator using DORA could modify a configuration file to include this information and replay the recorded execution with the modified configuration to generate the desired web server log.

However, doing this yields a log file with incorrectly truncated entries. This behavior was due to a previously unknown bug in Apache. In several places in code, Apache mistakenly assumes that a call to `write()` will either write the desired amount of bytes or fail, instead of checking the return value and calling `write()` until all the required bytes are written. We submitted a bug report and patch to Apache [1] which was accepted into the codebase.

apache-sec records an exploit of a heap overflow vulnerability that launches a denial of service attack against an Apache web server using only a handful of requests. By examining Apache's log, an experienced Apache developer will notice oddities in some of the requests, but will not have enough information to identify

Name	Description	Problem/Exploit Workload	Production Workload
apache-log	Apache 2.4.2 web server	Log format change (Apache Bug 53131)	`httperf 0.8` with 100KB web page
apache-sec	Apache 2.2.19 web server	DoS attack (CVE 2011-3192)	`httperf 0.8` with 100KB web page
exim	Exim 4.69 mail server	Privilege escalation (CVE 2010-4344)	Send 1000 1KB e-mail messages
mysql	MySQL 5.0.67 database server	Unauthorized access (CVE 2008-2079)	`sql-bench`
nginx	Nginx 0.8.14 web server	Crash server (CVE 2009-2629)	`httperf 0.8` with 100KB web page
proftpd	ProFTPD 1.3.0 ftp server	Crash server (CVE 2006-5815)	100 clients fetch 10MB file
redis	Redis 2.4.11 key-value store	Request with insufficient logging	`redis-benchmark` with 50 clients
squid	Squid 3.1.7 http proxy server	DoS attack (CVE 2010-3072)	`ab 2.3` with cached facebook.com
wget	wget 1.11.4 http client	Create arbitrary file (CVE 2010-2252)	100 requests to http://www.cnn.com

Table 2. Application scenarios

Name	Debugging Change	Replay Mutations
apache-log	Modify log format in configuration file	Add 1 `fstat()`, 1 `mmap()`, and 3 `write()` per request
apache-sec	Add print statements for debugging	Add 1 `fstat()`, 1 `mmap()` and then 2 `write()` per request
exim	Recompile with debugging options enabled	None
mysql	Add conditional print statements for debugging	Add 1 `fstat()`, 1 `mmap()`, 1 memory event, 12 `write()`
nginx	Change the config file to enable debug messages	Add 508 `write()`, delete 1064 syscalls
proftpd	Recompile with debugging options enabled	Add 1 `close()`
redis	Log erroneous client requests	Add 1 `open()`, at least 3 `write()` and 1 `close()` per request
squid	Save parsed requests to file	Add 1 `open()`, at least 10 `write()` and 1 `close()` per request
wget	Change language from Italian to Japanese	Replace 17 `write()`, 24 `mmap()`, 2 `open()`, and delete 5 syscalls

Table 3. Application modifications for debugging

the bug with the default logging settings. In particular, it would be helpful to have more information about the range headers of the requests. Using DORA, a developer can add print statements to Apache, replay, and recognize that the problem was due to incorrect handling of overlapping range headers. Five system calls were added for each request as a result of the additional print statements.

exim involves an exploit that crashes the mail server using a heap overflow vulnerability in a buggy string formatting function that allows attackers to execute arbitrary code. If this crash was recorded in the wild, it would be helpful to use GDB to analyze the program at the time of the crash. However, production servers are almost always optimized and compiled without debugging symbols. Thus, a traditional record-replay system would be unable to help. Using DORA, a developer can recompile the program, replay the exploit using the recompiled program, and hook GDB to the replayed program before it crashes. When investigating the stack trace, the nature of this attack becomes clear. No mutations were needed in the d-optimal replay, despite various memory layout changes to the program as a result of recompilation.

mysql involves an exploit which maliciously uses symlinks to elevate permissions to a database. By default, MySQL disables logging. Thus, a developer trying to discover how a malicious user gained access to a database will have no information about which commands were executed. Using DORA, the developer can modify the program to log executed commands, then replay the exploit using the modified program. This process can be repeated, allowing the developer to iteratively add print statements to different sections of the code and pinpoint the bug. To demonstrate this, we added enough print statements to identify the bug. DORA found the d-optimal replay, which had mutations of fourteen added system calls and a shared memory event.

nginx involves running Nginx, a high-performance HTTP server, and crashing one of its worker processes with a malicious HTTP request that uses a buffer underflow attack to execute arbitrary code. The default log does not show what actions were taken on each request, which makes debugging difficult. Using DORA, the developer can modify the configuration file to enable verbose logging and replay the exploit with the modified configuration. To generate verbose logs on a workload exhibiting the exploit, 508 `write()` calls were added and 1064 system calls were deleted.

proftpd records an exploit that crashes the FTP server by taking advantage of an off-by-one error to execute arbitrary code. As with the exim use case, GDB would be a helpful debugging tool, but a production server is unlikely to be compiled with debugging symbols. With DORA, a developer can recompile with the Makefile configuration for debugging, replay the exploit using the recompiled program, and hook GDB to the replayed program before it crashes. The resulting stack trace makes it easy to diagnose the problem. A replay mutation of adding 1 `close()` was needed to use the debugging configuration.

redis involves recording Redis, an in-memory key-value store often used in production applications as a caching layer on top of a general purpose database. This use case does not involve a specific bug but instead shows how a developer can add logging to Redis and replay this modification on the original recording, effectively turning on retroactive logging. Replay mutations of adding at least five system calls per request was needed. The number of additions varied based on the nature of the request. For instance, a malformed request triggered more logging than a proper request.

squid involves an exploit that crashes the Squid daemon by sending an empty Expect HTTP header parameter. The default request logging does not provide enough information about the header to determine the cause of the bug. Using DORA, a developer can modify the program to log each request to a file with the complete header information of the request, then replay a recording of the exploit with the modified program. This allows the developer to see that the requests which crash the server have empty header parameters and narrow down the bug to a very specific section of code. At least 12 system calls were added per request. The exact number varied depending on the type of request.

wget involves downloading a file from a malicious server that does a 301 redirect. A vulnerability in wget allows the server to choose the destination filename. Remote servers can create or overwrite arbitrary files and even execute arbitrary code by writing dot-

file in a home directory. Because this behavior depends on a live server behaving in a particular way, this issue may be difficult to reproduce and debug if it is not noticed immediately. Additionally, to demonstrate the robustness of DORA, we suppose that an Italian developer observes this behavior and wants to show it to a Japanese developer who cannot reproduce the results because the server is no longer available. Using DORA, the second developer can replay wget in a different language, enabling collaborative debugging across international borders and language barriers. While this scenario is tongue-in-cheek, the translation use case is novel and has interesting applications. The replay involved replacing 43 system calls and deleting 5 system calls.

8.2 Patch Validation

For each bug exhibited by the workloads listed in the third column of Table 2, we replayed the bug-inducing workload on the patched application to verify that the patch successfully fixed the bug. Table 5 lists the number of lines of code added and deleted for each patch and the mutations needed for each replay. Redis is not included; since it did not involve an application bug, no patch was necessary. Table 5 shows three interesting points.

First, DORA found the d-optimal replay even with substantial patches of over 400 lines of code changed. The replay mutations that were needed to find the d-optimal replay varied. Many involved executing different system calls, but others involved changes in signal delivery and shared memory accesses. This demonstrates DORA's ability to replay despite a broad range of application modifications so long as the core application semantics remain the same.

Second, we show that production workloads can be replayed using patched applications to verify that the patch does not introduce errors into those workloads. For each workload listed in the last column of Table 2, DORA recorded the workload using the unpatched application, then found a d-optimal mutable replay using patched versions of each application. We examined the output of each mutable replay and verified that the patches did not change application behavior when running the workload. We also compared each original recorded log with the log of the corresponding mutable replay to verify that the patches did not change the application execution in unexpected ways. System administrators could use this technique to test patches before deploying them to have more confidence that they will not break their production systems.

Third, Table 5 shows that the go live feature of the replayer can be used to validate patches even when a recorded exploit crashes a process. The exploit for Nginx caused a worker to crash, and the Squid exploit crashed the entire application. In both cases, DORA does not replay the original `SIGSEGV` and allows the applications to go live and handle new requests. Although a worker process crashed in the proftpd scenario, DORA did not go live because the proftpd master forks a new worker per connection and is resilient to worker crashes, allowing subsequent requests to be replayed without going live.

8.3 Release Upgrades

To demonstrate another use case of DORA, we took two server applications and recorded them running the benchmarks we used as production workloads as listed in Table 1. We then replayed those executions over a series of 15 release upgrades to verify that the workloads continued to function correctly across upgrades. DORA found the d-optimal replay in all of these cases.

apache-upgrade consists of a series of upgrades of Apache over an 8 month timeframe from 2.2.19 (May 21, 2011) to 2.2.22 (Jan 30, 2012). The upgrades involved changes of more than 5000 lines of code and 277 separate commits. Using DORA, we recorded version 2.2.19 running httperf, then replayed the recording with each subsequent version. We repeated this process for versions 2.2.20 and 2.2.21. DORA successfully replayed all of these application changes. This required various add and delete mutations of `read()` and `brk()` calls. Note that we also tried this experiment starting with Apache 2.2.18, but DORA was unable to replay from that version due to core library modifications that caused large shared memory layout changes between Apache 2.2.18 and 2.2.19.

Name	Recording Overhead	Storage Growth	Replay Speedup
apache-log	9.3%	31 KB/s	3.8x
apache-sec	4.8%	18 KB/s	1.9x
exim	4.3%	30 KB/s	7.2x
mysql	4.7%	9.6 KB/s	1.1x
nginx	9.7%	15 KB/s	2.2x
redis	2.6%	91 KB/s	1.3x
proftpd	4.1%	22 KB/s	2.6x
squid	8.2%	124 KB/s	1.2x
wget	2.2%	19 KB/s	11x

Table 4. Mutable replay performance

redis-upgrade consists of a series of upgrades of Redis over a 7 month timeframe from 2.4.1 (October 17, 2011) to 2.4.13 (May 2, 2012). The upgrades involved changes of more than 4000 lines of code in 137 separate commits. Using DORA, we recorded version 2.4.1 running redis-benchmark, then replayed the recording with each later version. We also repeated this experiment for version 2.4.2 and upgrades 2.4.3 to 2.4.13, version 2.4.3 and upgrades 2.4.13 to 2.4.4, and so on. DORA successfully replayed all of these application changes. They required add and delete mutations of 12 different system calls, including `open()`, `close()`, `read()`, `write()`, `mmap()`, `munmap()` and `time()` system calls.

8.4 Performance

To quantify the performance costs of using DORA, we measured the runtime overhead of recording and replaying the production workloads listed in the last column of Table 2. Table 4 shows the overhead of recording the production workload with the unpatched application, the storage growth rate of recording, and the speedup when replaying the recording with the patched application. Unless otherwise noted, default configuration options were used for all applications. The standard deviations for all measurements were negligible.

The recording overhead in all cases was less than 10% even for CPU-bound workloads designed to stress application performance. For example, Squid performance was measured with a fully cached web page, resulting in a CPU intensive workload. Even with these unfavorable workloads, the results indicate that DORA can be used in production systems with modest overhead.

Similarly, the time to replay the original recording on the original application was in all cases faster than the original execution; in one case, it was over an order of magnitude faster. This is because DORA can bypass blocking system calls that sleep. Since production servers are likely to sleep more and service requests less frequently than in our benchmarks, replay speedup will be much higher in practice.

While recording, the log was streamed through `gzip` before being persisted to disk. The storage growth rates ranged from 10 KB/s to 130 KB/s. These storage requirements are modest considering our workloads. DORA would take almost three months to fill a 1 TB drive at the worst of these rates, which makes it an affordable and practical solution.

Name	Patch LOC +/-	Replay Mutations
apache-log	39+, 39-	Add 1 `write()` per truncated log entry
apache-sec	292+, 154-	Add 1 `write()` and replace 1 `writev()` per request
exim	7+, 0-	Delete 18 syscalls, add 29 syscalls
mysql	170+, 60-	Add 29 `lstat()`, delete 79 syscalls, add 1 `write()`, delete 15 and add 8 memory events
nginx	9+, 5-	Delete 1 `write()` and replace 1 `writev()` per request, then go live on crash
proftpd	17+, 3-	Delete 694 syscalls, add 38 syscalls, delete a `SIGSEGV`
squid	38+, 33-	Delete a `SIGSEGV`, 1 `close()`, 1 `stat()`, 1 `write()`, then go live
wget	43+, 12-	Replace 9 syscalls among `stat()`, `write()`, `open()`, `utime()`

Table 5. Application patches tested against exploits

9. Related Work

Many record-replay approaches have been proposed to improve bug reproducibility debugging [2, 3, 7, 11, 18, 19, 21, 22, 26, 27], but none allows for mutable replay. Some approaches propose relaxing the requirement of deterministic replay for performance reasons. For example, ODR [2] proposes only ensuring that the output is deterministically replayed for replay debugging. This is quite different from mutable replay, in which the output may change due to application changes.

Some record-replay systems can support a form of deterministic replay that may differ in limited ways from the original recorded execution. Crosscut [8] can reduce the information recorded in a log so that, for example, sensitive information can be purged before replay. Our previous work on Scribe [16] replays a recorded application execution until a specified point, and then transitions to live execution instead of replaying the rest of the log. Our previous work on Racepro [17] detects process races due to dependencies in the ordering of system calls by recording an application execution to a log, identifying a pair of system calls that may be racy, truncating the log at the occurrence of the pair of system calls, inverting their order, and then replaying the truncated log with the reordered system calls to detect process races. However, Racepro only supports changes that reorder system calls and does not support changes in the middle of replay. None of these approaches supports mutable replay, but mutable replay could be useful for some of these systems. For example, Racepro could use mutable replay to avoid replay divergence and more effectively detect process races. Another race detection tool [19] uses the iDNA [3] record-replay framework and would also benefit from mutable replay.

A few record-replay systems allow new code to be run while replaying a recorded execution [7, 12]. However, this new code cannot have any side effects on the program. If a replay diverges due to new code, these systems must rollback to a point prior to the divergence for the replay to continue. In contrast, DORA allows replay to continue even after divergence; side effects due to new code are preserved. Moreover, unlike DORA, these other approaches prevent application developers from leveraging existing configurable application functionality and instead require that developers learn a new complex system. Because these other approaches work at a VM level, they are fundamentally limited in their abilities to perform mutable replay and support the kind of application changes supported by DORA. Finally, none of these other approaches work on multicore or multiprocessor systems.

A concept of mutable replay was mentioned as a part of DSF [28], a Java-only framework for implementing distributed algorithms. DSF recognized that existing replay approaches did not allow adding print statements for debugging. DSF requires that all applications to be written using its framework, requires modification to the applications, and is primarily simulation-based. Furthermore, DSF presents no algorithms or mechanisms for actually doing mutable replay, and presents no experimental results demonstrating the ability to do mutable replay. More recently, a study has assessed the potential utility of mutable replay on real patches [13], though no mutable replay system or results are presented. DORA presents the first system that achieves transparent mutable replay, requires no application modifications, and demonstrates experimentally that mutable replay can be used with real applications.

Alternative techniques have been proposed to help with patch validation, one use case of mutable replay. Band-aid patching [23] and delta execution [29] instrument patches to identify portions of an application that have changed, execute both the unpatched and patched code paths either serially or in parallel, and select the results from one path or merge the results from both paths. However, these approaches incur substantial performance overhead. Many simple patches cannot be handled by these approaches, such as simple changes to data structures. Unlike DORA, these approaches do not allow patch validation on a recorded bug or offline patch validation on a production workload. Furthermore, they are designed for patch validation only and are not effective for debugging.

Self-healing systems have been proposed which record the occurrence of a bug, then automatically generate and apply a patch as a temporary fix to the problem [24]. DORA is complementary to these systems and can be used to verify that a generated patch successfully fixes problems that occurred in the original workload.

Finding a mutable replay has some similarities to the edit distance and longest common subsequence problems, which have applications to approximate string matching and bioinformatics. In those problems however, both sequences being used for matching are known in advance. In contrast, mutable replay must match a known execution log with an execution sequence that is not known in advance, so these algorithms cannot be directly applied.

10. Conclusions and Future Work

DORA introduces the concept of mutable record-replay and is the first transparent mutable record-replay system. It enables, for the first time, a recording of an application execution to be replayed using a modified version of the application for a large class of application changes. This is made possible by the use of lightweight operating system mechanisms to record and replay without imposing unnecessary timing and ordering constraints. DORA introduces an explorer that directs the replay mechanism to identify a mutable replay of the modified aplication that minimizes differences with the original unmodified application execution.

Our experimental results on a Linux prototype demonstrate that mutable replay is feasible across a wide range of real-world applications and application changes which can reach thousands of lines of code, even without support for major changes to core application semantics. We show that mutable replay is useful for enabling common debugging techniques not possible with previous record-replay systems. We also show that mutable replay enables validation of security patches against both exploits and production workloads. This is all accomplished without requiring source code modifications and with low recording overhead, enabling usage on

production systems. These results demonstrate that mutable replay has the potential to enable new techniques for debugging and patch testing and validation, which can lead to substantial improvements in software reliability and developer productivity.

We hope to explore a number of directions in future work. While we have explored a few ways in which mutable replay can be used in multicore debugging and patch validation, DORA provides a foundation for exploring other uses of mutable replay as well. We have evaluated a particular cost function and exploration algorithm for mutable replay, but much more can be done to consider alternatives that may be particularly suited for other use cases. Furthermore, it would be beneficial to perform instrumentation at a higher level: if runtime systems like the Java Virtual Machine or the Ruby MRI were instrumented to work with DORA, mutable replay could be an even more effective tool to reproduce, diagnose, and fix software bugs for Web and mobile applications.

Acknowledgments

Oren Laadan and Dan Tsafrir provided helpful comments on earlier drafts of this paper. This work was supported in part by an IBM Faculty Award and NSF grants CNS-1018355, CNS-0905246, and CCF-1162021.

References

[1] Apache Bug 53131. `https://issues.apache.org/bugzilla/show_bug.cgi?id=53131`.

[2] G. Altekar and I. Stoica. ODR: Output-Deterministic Replay for Multicore Debugging. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, Nov. 2009.

[3] S. Bhansali, W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, and J. Chau. Framework for Instruction-level Tracing and Analysis of Program Executions. In *Proceedings of the 2nd International Conference on Virtual Execution Environments (VEE '06)*, June 2006.

[4] E. W. Biederman. Multiple Instances of the Global Linux Namespaces. In *Proceedings of the Linux Symposium*, July 2006.

[5] T. C. Bressoud. TFT: A Software System for Application- Transparent Fault Tolerance. In *Proceedings of the 28th International Symposium on Fault-Tolerant Computing (FTCS '98)*, June 1998.

[6] T. C. Bressoud and F. B. Schneider. Hypervisor-Based Fault Tolerance. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, Dec. 1995.

[7] J. Chow, T. Garfinkel, and P. M. Chen. Decoupling Dynamic Program Analysis from Execution in Virtual Environments. In *Proceedings of the USENIX Annual Technical Conference (USENIX '08)*, June 2008.

[8] J. Chow, D. Lucchetti, T. Garfinkel, G. Lefebvre, R. Gardner, J. Mason, S. Small, and P. M. Chen. Multi-stage Replay with Crosscut. In *Proceedings of the 6th International Conference on Virtual Execution Environments (VEE '10)*, Mar. 2010.

[9] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling Intrusion Analysis Through Virtual-Machine Logging and Replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Dec. 2002.

[10] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution Replay of Multiprocessor Virtual Machines. In *Proceedings of the 4th International Conference on Virtual Execution Environments (VEE '08)*, Mar. 2008.

[11] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang. R2: An Application-Level Kernel for Record and Replay. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI '08)*, Dec. 2008.

[12] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. Detecting Past and Present Intrusions through Vulnerability-Specific Predicates. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, Oct. 2005.

[13] I. Kravets and D. Tsafrir. Feasibility of Mutable Replay for Automated Regression Testing of Security Updates. In *Workshop on Runtime Environments, Systems, Layering and Virtualized Environments (RESoLVE)*, March 2012.

[14] O. Laadan and J. Nieh. Transparent Checkpoint-Restart of Multiple Processes on Commodity Operating Systems. In *Proceedings of the 2007 USENIX Annual Technical Conference*, June 2007.

[15] O. Laadan, R. A. Baratto, D. Phung, S. Potter, and J. Nieh. DejaView: A Personal Virtual Computer Recorder. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, Oct. 2007.

[16] O. Laadan, N. Viennot, and J. Nieh. Transparent, Lightweight Application Execution Replay on Commodity Multiprocessor Operating Systems. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '10)*, June 2010.

[17] O. Laadan, N. Viennot, C.-c. Tsai, C. Blinn, J. Yang, and J. Nieh. Pervasive Detection of Process Races in Deployed Systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, Oct. 2011.

[18] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Transactions on Computers*, 36(4), Apr. 1987.

[19] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically Classifying Benign and Harmful Data Races Using Replay Analysis. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI '07)*, June 2007.

[20] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Dec. 2002.

[21] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. PRES: Probabilistic Replay with Execution Sketching on Multiprocessors. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, Oct. 2009.

[22] Y. Saito. Jockey: a User-Space Library for Record-Replay Debugging. In *Proceedings of the 6th International Symposium on Automated Analysis-Driven Debugging (AADEBUG '05)*, Sept. 2005.

[23] S. Sidiroglou, S. Ioannidis, and A. D. Keromytis. Band-aid Patching. In *Proceedings of the 3rd workshop on on Hot Topics in System Dependability (HotDep '07)*, June 2007.

[24] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A. D. Keromytis. ASSURE: Automatic Software Self-healing Using REscue points. In *Proceedings of the 14th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '09)*, Mar. 2009.

[25] J. H. Slye and E. Elnozahy. Supporting Nondeterministic Execution in Fault-Tolerant Systems. In *Proceedings of the 26th International Symposium on Fault-Tolerant Computing (FTCS '96)*, June 1996.

[26] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: A Lightweight Extension for Rollback and Deterministic Replay for Software Debugging. In *Proceedings of the USENIX Annual Technical Conference (USENIX '04)*, June 2004.

[27] D. Subhraveti and J. Nieh. Record and Transplay: Partial Checkpointing for Replay Debugging Across Heterogeneous Systems. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '11)*, June 2011.

[28] C. Tang. DSF: A Common Platform for Distributed Systems Research and Development. In *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware (Middleware '09)*, Nov. 2009.

[29] J. Tucek, W. Xiong, and Y. Zhou. Efficient Online Validation With Delta Execution. In *Proceedings of the 14th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '09)*, Mar. 2009.