Abstract
Reproducing errors of multithreaded programs is very challenging due to many intrinsic and external non-deterministic factors. Existing RnR systems achieve significant progress in terms of performance overhead, but none target the in-situ setting, in which replay occurs within the same process as the recording process. Also, most cannot achieve identical replay, which may prevent the reproduction of some errors.

This paper presents iReplayer, which aims to identically replay multithreaded programs in the original process, under the “in-situ” setting. The novel in-situ and identical replay of iReplayer makes it more likely to reproduce errors, and allows it to directly employ debugging mechanisms (e.g., watchpoints) to aid failure diagnosis. iReplayer implements multiple methods to reduce its performance and support identical replay. Currently, iReplayer only incurs around 3% performance overhead, which allows it to be always enabled in the production environment. iReplayer enables a range of possibilities, and this paper presents three examples: two automatic tools for detecting buffer overflows and use-after-free bugs, and one interactive debugging tool that is integrated with GDB.

Keywords Record-and-Replay, Identical Re-execution, Dynamic Analysis, Multithreaded Programs

1 Introduction
Multithreaded programs contain intrinsic non-deterministic factors that may affect the schedule and results of different executions. Thus, it is very challenging to reproduce errors in multithreaded programs. Record-and-Replay (RnR) systems record non-deterministic events of the original execution in order to assist the replay [63]. Typically, they record the order of synchronizations and the results of certain system calls from the original execution, and then reproduce them during the re-execution [59]. Some RnR systems even record the order of memory accesses [12], or utilize offline analysis to infer the order of memory accesses inside the execution [5, 35, 36, 44]. However, existing RnR systems have two shared shortcomings, in addition to their specific problems, as described in Section 7.

First, existing RnR systems typically reproduce the execution in a different process, starting from the beginning. They could possibly achieve better diagnostic capability, since they can access all information from the entire execution [6]. However, the replay-from-beginning is difficult to employ in reality. (1) it is much more difficult to build an automatic tool on top of it, since the tool should connect a known failure to a variable with a different address in the re-execution, due to the fact that most RnR systems cannot reproduce the execution; (2) It is harder to employ existing debugging mechanisms directly, such as watchpoints, if memory addresses are different. (3) It may take a long time (e.g., up to days) to reproduce the entire execution, such as for server programs or chip emulation software. This fact may result in inefficient debugging, since programmers may spend a significant amount of time waiting for the occurrence of a problem [3].

Figure 1. A null reference problem in original execution.

Second, all existing RnR systems (except RR [52, 57]) cannot identically reproduce the recorded execution, as they do not guarantee the same system states, such as process/thread IDs and file descriptors [5, 15, 31–33, 35, 44, 50, 51, 59, 64, 70], the same results of system calls (e.g., gettimeofday) [34, 35, 46, 49], or have different memory layouts [5, 15, 31–33, 35, 41, 44, 50, 51, 59, 64, 70]. Therefore, it is impossible to reproduce some types of bugs: (1) Bugs related to memory layout. Figure 1 shows an example of a crash due to dereferencing a null pointer overwritten by a buffer overflow bug. A different memory layout, where an integer (not the pointer) is allocated immediately after the overflowing object, may hide this crash during re-executions; (2) Bugs depending on system states, such as thread IDs, file descriptors, or memory addresses, may not be reproducible when these system states are not preserved during the original execution. In fact, some real systems were designed to utilize these system states explicitly. For instance, the Hoard allocator assigns heaps to each thread based on the hashing of their thread ID [10], and some hash tables use object addresses as their keys [38]. For these systems, existing RnR systems may not reliably reproduce latent bugs, since both thread IDs and memory addresses are typically different between distinct executions. On the other hand, RR runs multithreaded programs on a single thread, which may hide many concurrency bugs and cannot employ the potential of multicore hardware [52, 57].

This paper presents iReplayer, a novel system that targets to in-situ and identically replay multithreaded programs,
which has the following significant differences from existing RnR systems.

Firstly, iReplayer designs a novel in-situ replay technique that always replays the last-epoch execution within the same process as the original execution. Replaying in the same process (in-situ replay) makes it more likely to reproduce identical system states and more bugs. This “in-situ” replay is different from existing online replay, such as Rx [61], Triage [68], and DoublePlay [70], where the replays of these systems actually occur in a process different from the recorded one. Currently, iReplayer only replays the last-epoch execution by default. However, it is especially suitable for identifying bugs. Based on recent studies [6, 27, 61], most bugs have a very short distance of error propagation, which indicates a root cause may be located shortly prior to failures. Replaying the last epoch of execution also helps improve the debugging efficiency of programmers, since it avoids significant time spent waiting for problems to appear.

Secondly, iReplayer aims for identical re-execution that strictly preserves all system states, results of system calls, the order and results of synchronizations, and the same memory allocations/deallocations of the original execution, even for racy applications. Re-execution in the same process as the original execution helps preserve system states, such as process IDs. Additionally, iReplayer handles system calls specially, delays the reclaiming of threads in order to maintain the state of memory mappings and IDs for each thread, and employs a custom memory allocator to manage the application heap similarly across multiple executions, as described in Section 2.2. Based on our evaluation, iReplayer can identically reproduce all evaluated applications that do not have ad hoc synchronizations inside. Programs using ad hoc synchronizations are considered to be error-prone and should be avoided [72].

Thirdly, iReplayer only imposes around 3% recording overhead, which is sufficiently low for the deployed environment. iReplayer utilizes multiple approaches to reduce its logging overhead: (1) iReplayer designs a novel data structure that allows it to record the local-order of synchronizations efficiently, while still ensuring identical replay. This data structure can easily support identical replay as well. (2) It prevents significant logging overhead caused by logging every memory access of the original execution, a common approach to handle race conditions [5, 12]. Instead, iReplayer only handles race conditions in the replay phase, similar to existing work [43, 50]. (3) iReplayer designs an indirect level for synchronization variables to further reduce performance overhead. More can be seen in Section 3.2.

The identical and efficient in-situ re-execution provided by iReplayer enables a range of possibilities: (1) It will enable evidence-based approaches which can diagnose root causes of multiple types of problems, such as memory errors, segmentation faults, aborts, assertions, or system intrusions; (2) It will enable evidence-based error prevention, such as memory errors, deadlocks, etc. (3) It can be always enabled, so that programmers may utilize it to identify root causes in case of abnormal exits. The paper also shows three applications of iReplayer: that both normal users and experts can benefit from: two tools for detecting buffer overflows and use-after-frees, and one interactive debugging tool that can be connected with GDB.

Overall, this paper makes the following contributions:

An in-situ record-and-replay technique for multithreaded programs: iReplayer proposes an in-situ RnR system, in which replay occurs in the same process as the original execution.

An identical replay technique: iReplayer supports identical replay for programs without ad hoc synchronizations, in which all system states, results of system calls, order of synchronizations, and memory uses of the re-execution are the same as the original one.

Novel implementation to reduce overhead: iReplayer proposes a novel data structure that enables the identical replay, but with low recording overhead. This data structure also supports the checking of divergence easily.

A practical system combining low recording overhead, identical replay, and convenience: iReplayer is a software-only solution with negligible recording overhead, only 3% on average. iReplayer is a drop-in library that runs entirely within the user space, and does not require complex nonexistent hardware, customized OS, or the modification of programs.

Multiple promising applications: Two automatic tools of identify root causes of buffer over-writes and use-after-free errors, and one interactive debugging tool (connecting with GDB) were developed to show the usefulness of iReplayer. 

[[Change something to the first one]]

Outline:

The remainder of this paper is organized as follows. Section 2 gives an overview of our approach, including the challenges of implementing multithreading support. After that, Section 3 presents the detailed implementation, and Section 4 discusses several applications built on iReplayer. Section 5 presents experimental results and limitations are discussed in Section 6. Finally, Section 7 reviews related work, and Section 8 concludes.

2 Overview

This section provides an overview of iReplayer, and the major challenges of supporting in-situ and identical replaying of errors in multithreaded applications.

2.1 Overview of Execution

iReplayer divides the entire execution into multiple epochs, based on irrevocable system calls (defined in Section 2.2),
The start of a program is considered the start of the first epoch, and termination (either a normal or abnormal exit) will serve as the end of the last epoch. iReplayer marks its initialization function with the constructor attribute, which allows it to initialize its custom heap, install signal handlers, and prepare internal data structures for recording, before entering the main routine. During initialization, iReplayer also identifies the range of global and text segments for the application, as well as any libraries, by analyzing the /proc/self/maps file. This information will be utilized for checkpointing in the original execution, or for preparation for re-execution. iReplayer starts the first epoch just before the application enters its main function, and after initialization has been completed.

2.2 Challenges for In-situ and Identical Replay

The major challenge of iReplayer is to ensure identical replay in the in-situ setting. More specifically, identical replay requires the faithful reproduction of the order and results of synchronizations, the order of memory accesses, the states following system calls, and the memory layout of the original execution, as described in the following.

2.2.1 Ensuring Identical Synchronizations

To ensure the same order of explicit synchronizations, we record the order of synchronizations during the original execution, then replay this order in re-executions [59]. However, the greatest challenge is to achieve efficient recording, which is further described in Section 3.2. Some applications also rely on the results of synchronization functions, such as try locks or barrier waits. Thus, iReplayer also records the return values of these synchronizations, as described in Section 3.2. Currently, iReplayer does not handle ad hoc synchronizations, which are considered to be harmful and should be avoided if possible [72]). If ad hoc synchronizations are changed to explicit synchronizations (e.g., pthread_mutex_lock), iReplayer can replay the application identically. Furthermore, we proved this in canneal, as described in Section 5.2.

2.2.2 Ensuring Identical Order of Memory Accesses

There are two types of memory accesses in a multithreaded application, namely thread-private and shared accesses. The order of thread-private accesses is determined by the instruction execution order of a single thread. Therefore, it does not require special handling in order to be identically replayed. For shared accesses, if they are properly protected by explicit synchronizations, their order is determined by the order of
the explicit synchronizations, which has been handled, as discussed above. Otherwise, they may cause race conditions.

**Handling race conditions:** iReplayer does not record racy accesses initially, since that is too expensive [12], but only the events of synchronizations and system calls. Instead, it handles the race condition inside the replay phase, which avoids significant recording overhead for common cases in which programs do not expose race conditions. During replay, it will check for divergence from the recorded events, which must be caused by race conditions (except for ad hoc synchronizations, which iReplayer does not support). Therefore, iReplayer utilizes multiple replays to search for a matched schedule. When iReplayer finds a schedule that matches the recorded events, it assumes that the replay is identical to the original execution, and will stop searching. Although there is some chance that the replay is still different from the original execution, it is very unlikely to occur or is caused by benign races that users do not care about. This general idea is inspired by existing work [43, 50], but with some difference. Lee et al. utilizes a single-threaded execution to replay multithreaded applications, which requires a large number of searches as self-acknowledged [43]. Castor although discusses about the option of using re-running with partially ordered replay, but does not include any implementation details or evaluations on it [50].

### 2.2.3 Ensuring Identical System Calls

Existing RnR systems record the results of system calls to ensure the same states during replay [59, 63]. For instance, the recorded results of gettimeofday() can be utilized in the replay phase to supply the system call results without actually invoking this system call. However, they do not handle all system calls. They typically do not handle the invocation of pthread_create(), which is not guaranteed to create a thread with the same thread ID and stack as the original. If a program’s execution relies on this information, such as the Hoard allocator mentioned in Section 1, then it is impossible to ensure identical replay.

The in-situ setting imposes some additional challenges for certain file-related invocations. For instance, we assume a sequence of file-related system calls, such as {open(1), close(1), fopen(2)}, where fopen(2) will return the same file descriptor as open(1). Since fopen(2) already occupies the file descriptor of open(1), without returning it to the OS, then open(1) will return a different file descriptor in the re-execution. Similar results may occur for the munmap system call. As another example, fread invokes the mmap system call internally to allocate buffers, which cannot be reproduced identically due to the ASLR mechanism [16]. iReplayer invokes the setvbuf API to set up the stream buffer explicitly to avoid this issue.

iReplayer classifies system calls into five categories, similar to DoubleTake [47]. **Repeatable system calls** always return the same results, e.g. getpid(), without changing internal system state. They require no special handling. **Recordable system calls** return different results when invoked during re-executions. Examples include gettimeofday() and socket reads/writes. iReplayer records the results and return values of these system calls during the original execution, so that the same results can be used during replay without performing actual invocations. **Revocable system calls** modify system states, but the operations can be reproduced by recovering initial states (e.g. file positions) prior to re-execution. These mainly include file-related reads/writes. Although their results can be recorded, this may impose substantial recording overhead. However, if a write changes the data after invoking lseek, then it is not able to reproduce any read prior to the lseek. Therefore, iReplayer treats lseek (with changing positions) as irrevocable system calls. **Deferrable system calls** irrevocably change system states, but can be safely delayed. These system calls are very important for identical re-execution in an in-situ setting. Examples include munmap and close. iReplayer delays these system calls until the next epoch, when there is no need for re-execution. Note that delaying close() may result in the number of open files exceeding the default limit; therefore, iReplayer increases this limit during initialization. However, only privileged users are permitted to do so. Therefore, iReplayer may fail to run some applications, if this limit is exceeded. **Irrevocable system calls** irrevocably change system states, and cannot be rolled back safely or deferred. A thread encountering such system calls will coordinate to close the current epoch. For example, execve and fork are examples of these system calls.

This classification has a significant impact on performance. Although iReplayer could treat every system call as irrevocable, this would create a large number of epochs, and significantly increase the overhead caused by stopping, checking-pointing, and cleaning. Thus, irrevocable system calls are eliminated as much as possible. Some system calls are further classified based on their input parameters. For instance, the fcntl system call with the F_GETOWN flag is treated as a repeatable system call, while it will be treated as a recordable system call when used with the F_DUPFD flag.

### 2.2.4 Ensuring Identical Heap Layout

Memory allocation is a source of non-determinism in multi-threaded applications. First, the OS may randomize memory uses due to the ASLR mechanism [16], which will return different ranges of addresses upon mmap and sbrk system calls. Second, multiple threads may compete with each other for memory space. To ensure an identical memory layout, iReplayer isolates its internal memory uses from those of the application, adapts a “per-thread heap” so that memory allocations inside the same “heap” completely depend on the program order, and controls interactions with the outside
3 Implementation

This section describes the implementation, organized by phases, as shown in Figure 2.

3.1 Epoch Begin

At epoch begin, the major task is to checkpoint the states of the execution in order to support re-execution. If the epoch is not the first one, some housekeeping operations should be completed prior to checkpointing. In a multithreaded environment, a thread (typically the coordinator thread) is responsible for the housekeeping operations.

Housekeeping operations typically involve the removal of records from the previous epoch, such as the list of system calls and synchronizations. As described in Section 2.2.3, some system calls are delayed, such as close and munmap, and will be issued at this time. The cached data for closed sockets will be removed, and joined threads will be reclaimed.

After this, iReplayer checkpoints the states shared by all threads: the memory states, and positions of open files (see Section 3.2). Checkpointing memory states is performed by copying all writable memory to a separate block of memory, such as the heap and globals for both the application and its dynamically-linked libraries. This checkpoint is different from existing work [42, 61, 68, 70] to assist in-situ replay, which does not require changes to the underlying operating system. This fact improves its practicality. iReplayer utilizes a global hash table to store the file positions of all open files, which will be updated by invoking lseek at the epoch beginning.

Afterwards, all other threads are woken up, including threads waiting on conditional variables, barriers, and thread joining, so that they can check their own per-thread states. Per-thread states include the stack and per-thread hardware registers. iReplayer invokes getcontext to record the state of per-thread hardware registers.

3.2 Original Execution

During the original execution, iReplayer mainly handles system calls and synchronizations, and deals with memory allocations and deallocations as discussed in Section 2.2.4, in order to support in-situ and identical replay. iReplayer utilizes the following mechanisms to reduce recording overhead.

First, iReplayer designs a novel data structure (as shown in Figure 4) to store synchronization and system call events, which preserves the order of events in the same thread and across multiple threads. This data structure records the local order of synchronizations. Each event is recorded in its per-thread list initially, then will be added into the corresponding per-variable list. The per-thread list records the order of different events inside this thread, and the per-variable list records the temporal order of all threads’ activity on a specific synchronization variable. For the example shown in Figure 3,
the corresponding order will be recorded as in Figure 4. For instance, lock1’s per-variable list will track that lock1 is first acquired by Thread1, and then by Thread2.

This novel data structure is very efficient in recording. There is no need for additional synchronization when an event is added to its per-thread list, since no other threads will concurrently access this per-thread list. Adding an event to its per-variable list also requires no additional synchronization, since the addition is performed under the protection of the original lock. iReplayer further reduces its performance overhead by pre-allocating a specified number of entries for per-thread lists, which does not require additional memory allocations during recording. When all entries are exhausted, it is time to stop the current epoch and start a new epoch.

This data structure assists the identical re-execution of synchronizations as well, without the need of global order. Different synchronizations inside the same thread will always have the same order that determined by its program logic, and the per-variable list ensures that multiple threads will perform synchronizations in the recorded order. It is convenient for checking the divergence of re-executions: each thread is only required to check whether the next synchronization event or system call is the same as the next one in their per-thread list. If not, this is indication of a divergence, and iReplayer will immediately invoke re-execution in order to search for a matched schedule. More details are described in Section 3.5.

Second, iReplayer employs an indirect level for recording the events on each synchronization variable. Naively, we could maintain a global hash table to track the mapping between each synchronization variable and its own per-variable list. However, during development, this method was found to impose up to $4 \times$ performance overhead for applications with a large number of synchronizations, e.g. fluidanimate of PARSEC [13]. Instead, iReplayer allocates a shadow synchronization object from its internal heap (to avoid interfering with the application’s memory uses), and saves a pointer to this shadow object within the first word of the original synchronization object. This shadow object includes the real synchronization object and the header of its per-variable list. This mechanism allows for quickly obtaining per-variable lists with few operations. Allocation of these shadow objects is performed on initialization of synchronization objects, or upon the first lock acquisition, if the mutex was not initialized explicitly.

Third, iReplayer delays the handling of race conditions until the replay phase, which avoids recording the order of each memory access. The pros-and-cons of this idea have been described in Section 2.2.

Last but not least, treating file reads/writes as revocable system calls also helps reduce significant recording overhead.

### 3.2.1 Supporting Synchronizations

iReplayer supports a range of synchronization primitives, including thread creations, various forms of mutex locks, conditional variables, barriers, signals, and thread joins.

**Thread creation, destruction, and joins:** iReplayer intercepts pthread_create function calls, and passes an internal function to the actual system call. This allows iReplayer to initialize thread-related data, checkpoint the state prior to the execution, and handle future thread exits. iReplayer does not allow concurrent thread creations in order to guarantee the same order during re-execution, where the creation of threads is protected with a global mutex.

iReplayer keeps threads alive (without exiting) until the next epoch in order to preserve system states, such as thread IDs and stacks. In order to keep threads alive while allowing programs to proceed as usual, iReplayer emulates thread joining by using a thread-specific conditional variable and a status field. For a joinee thread that is typically joined by its parent, it checks this status field upon exit. If the parent thread has not yet joined on it, the status will be set to “joinable”, which may then be changed to “joined” with a subsequent join operation. Otherwise, the child thread wakes its joiner immediately. For both cases, the child thread is waiting on an internal thread-specific conditional variable (and thus kept alive), awaiting notification that it should either roll back or exit.

**Mutex locks:** For mutex locks, iReplayer records the order and return values of lock acquisitions using the data structures shown in Figure 4. For mutex try-locks, iReplayer also records the return value within per-thread lists, but only adds successful acquisitions into per-variable lists.
iReplayer: In-situ and Identical Record-and-Replay for Multithreaded Applications

Conditional variables: A cond_wait is treated as a mutex release followed by a mutex acquisition (when woken up). iReplayer records the wake-up events of conditional variables, similar to lock acquisitions. Since other threads may close the current epoch during waiting, every thread should update its status and record the conditional variable prior to waiting. After being woken up, a thread either proceeds as normal, or performs a re-execution or checkpointing. iReplayer does not record the order of events, such as cond_signal and cond_broadcast, as these are typically regulated by mutex locks.

Barriers: During barrier waiting, a thread may wait inside the kernel until the required number of threads have reached the same barrier. However, every thread must be woken up when the last epoch is going to be re-executed. However, there is no way to wake a thread waiting on the barrier until the required number of threads have reached the same barrier. To solve this issue, iReplayer re-implements the barrier using the combination of a mutex and a conditional variable, as it is easy to wake a thread waiting on a conditional variable by signal. Accordingly, the initialization of barriers is intercepted in order to allocate the mutex locks and conditional variables, then store their associated pointers in the first word of the barrier variable (similar to a mutex lock).

iReplayer does not record the order of entry into a barrier, since a thread that is waiting on a barrier will not change the status of the program. However, iReplayer records the return value of every barrier wait, since some applications rely on this value to determine the last thread entering into the barrier.

3.3 Epoch End

At epoch end, the coordinator thread is responsible for safely stopping the other threads and closing the current epoch. It is impossible to checkpoint a multithreaded program correctly and replay identical consequently, when multiple threads continue executing and changing states. Therefore, iReplayer takes the “stop the world” approach employed by garbage collection [14] to stop the execution. Stopping an epoch safely is unique to the in-situ setting, and has several challenges, as described below.

Challenge 1: What is the basic idea to stop other threads? A naive approach is to stop the threads using asynchronous methods, like signals. However, this approach was found to be very unreliable during our development. If the threads are waiting inside the kernel due to synchronizations (such as barrier_wait) or system calls, it is impossible to stop them reliably, and cleanly roll them back. iReplayer employs a synchronized method instead: before any synchronization or system call invocation, every thread checks whether a coordinator thread has requested to stop the current epoch. If so, the current thread will wait on its internal conditional variable, and mark its state as “stopped”. Therefore, there are no threads in the middle of system calls, except those waiting on different synchronizations.

Challenge 2: How to stop threads waiting on synchronizations, such as lock acquisitions or conditional variables? Threads waiting on conditional variables are considered to be in unstable states, since other threads may wake them up at any time. For those threads, iReplayer continues checking their states until all other active threads have reached their stable “stopped” states. Then, the epoch is safe to close, even when some threads are still waiting on conditional variables, since no other threads will send signals to wake these waiting threads. iReplayer also handles threads waiting on the acquisition of mutex locks: the actual holder of a mutex will release its lock temporarily before it stops, so that the waiter can acquire the lock and stop stably. iReplayer guarantees the lock will be returned to the original holder when the program proceeds as normal, without causing any atomicity violations.

When all other threads (except the coordinator thread) have been stopped in this way, the current epoch is closed. The coordinator thread should confirm the status of other live threads one-by-one. At epoch end, the coordinator thread proceeds to check whether a replay should be performed. If evidence of a program error exists (as shown in Section 4), or instructions have been received from the user, all threads are rolled back to the last checkpoint and re-executed from there. Otherwise, all threads proceed to the next epoch as normal (discussed in Section 3.1). The coordinator thread directs all of these operations.

3.4 Preparing for Re-execution

This section describes how iReplayer prepares for re-execution, and recovers the state of the program to that of the last checkpoint (called “rollback”). iReplayer prepares the re-execution in the following steps.

Firstly, the coordinator thread restores the memory of the heap and global sections, for both the application and all shared libraries, by copying the stored memory back to its corresponding locations.

Secondly, iReplayer ensures all per-thread lists and per-variable lists of synchronization events are pointing to their first recorded entry, which can be implemented very efficiently by simply changing pointers. For this purpose, iReplayer also maintains a list to track the mapping between synchronization variables and their shadow objects, so that every shadow object and their own per-variable lists (having the order of synchronizations) will be restored reliably, in order to guarantee identical replay.

Thirdly, iReplayer recovers the file positions of all open files from the last epoch, by invoking the lseek API directly with the SEEK_SET option on every file descriptor. In the end, the coordinator instructs other threads to roll back their own stacks and contexts themselves. (1) Threads waiting on conditional variables or barriers should first be...
woken. Threads created during the last epoch should wait for notification from their parents for re-execution, after their corresponding thread-creation events have transpired. (2) Rolling back the stack should be performed very cautiously, since the stack to be recovered could be larger than the current stack. Then, copying the stack directly might overwrite live values on the current stack, which can cause a program to behave abnormally. Basically, iReplayer forces such threads to use temporary stacks before copying, then switch back to their original stack after the copy has completed. (3) Some applications contain un-initialized reads that may access stack variables beyond the stack of the last checkpoint. Thus, iReplayer should zero out areas above the stack of the last checkpoint, in order to ensure identical replay. (4) If the rollback was caused by a program fault, such as SIGSEGV, iReplayer cannot perform the rollback directly inside the signal handler, which is using the kernel stack. iReplayer changes the IP pointer of the interrupt context to point to a custom function, such that the rollback can be performed in this function, following the signal handler. (5) In the end, each thread calls the setcontext API to restore its hardware registers, and begins re-execution immediately thereafter.

3.5 Re-executions
iReplayer’s re-execution has three goals. First, it should identically reproduce the original execution. Second, it should check for possible divergence caused by race conditions. Third, it should handle signals triggered by watchpoints for applications, as described in Section 4.

3.5.1 Repeating Original Execution
To achieve identical re-execution, iReplayer handles system calls correspondingly (see Section 2.2.3), repeats memory allocations and deallocations as described in Section 2.2.4, and repeats the recorded order of synchronizations.

As described in Section 3.2 (and shown in Figure 4), iReplayer introduces per-variable lists to link synchronizations across multiple threads. iReplayer utilizes a conditional variable and a lock to control the order of synchronizations in re-execution phases. The basic rule is listed as follows: whenever the first event of a per-variable list (e.g., a lock) is also the first event of its corresponding per-thread list, the current thread can proceed. Otherwise, the current thread should wait until previous synchronizations on this variable have transpired.

iReplayer utilizes a global mutex to control the order of thread creation. During re-execution, the parent thread waits for its turn to proceed, then notifies the corresponding child thread (waiting on their internal conditional variable) to proceed immediately. It skips the actual thread creation, since all threads were kept alive. This fact guarantees the same thread ID and stack for each thread. Other synchronizations are discussed in Section 3.2.

3.5.2 Checking Divergence
iReplayer checks for divergence from the recorded order for two types of events, system calls and synchronizations, using the data structure described in Section 3.2. For system calls, iReplayer confirms whether or not they are expected by comparing with the recorded events. For synchronizations, iReplayer confirms whether the address and type of synchronization is expected. As described before, any divergence from the recorded order can be only caused by unknown race conditions. Then, iReplayer utilizes multiple replays to search for a matched schedule. Currently, iReplayer supports an unlimited number of replays. Any divergence will trigger a new re-execution immediately. When the divergence is detected, the current thread immediately becomes the coordinator. It stops other threads, as described in Section 3.3, then invokes the normal rollback procedure after all threads have been reliably stopped (Section 3.4). Re-execution can be invoked multiple times, until there is no divergence from the original execution. Note that since iReplayer replays the original execution within the same process, with multi-threaded support; generally, it takes very few re-executions to find a matched schedule, as evaluated in Section 2.

4 Applications
This section presents three example applications built on top of iReplayer: two automatic tools for detecting heap buffer overflows and use-after-free memory errors, and one debugging tool integrating with GDB. The ideas of detecting memory errors are adopted from DoubleTake [47]. These applications exemplify the usefulness of an in-situ and identical record-and-replay system.

4.1 Heap Overflow
A heap buffer overflow occurs when a program writes outside the boundary of an allocated object. To aid in error discovery, iReplayer places canaries (e.g., known random values) adjacent to allocated objects in the original execution, a mechanism first introduced by StackGuard [19]. An overflow will corrupt the canary value, which can then be detected at the end of each epoch. Any overwritten canary is incontrovertible evidence that a buffer overflow has occurred. iReplayer uses a bitmap internally to record the placement of canaries.

After the discovery of an overflow, iReplayer immediately triggers a re-execution to locate the exact instructions responsible for the overflow. Before re-execution, iReplayer installs a watchpoint at every address with a corrupted canary. This can identify root causes of four buffer overflows in one re-execution. If applications have more than four bugs in one epoch, which is
very unlikely in deployed software, iReplayer may invoke multiple replays in order to identify root causes for all bugs.

4.2 Use-after-free

Use-after-free errors occur whenever an application accesses memory that has previously been deallocated, and has possibly been re-allocated to other live objects. A use-after-free error may lead to an immediate SIGSEGV fault, the corruption of data, or other unexpected program behavior.

To detect use-after-free problems, iReplayer delays the re-allocation of freed objects by placing them into a quarantine list, an idea originally developed by AddressSanitizer [65]. iReplayer fills the first 128 bytes of freed objects with canary values. These freed objects are released from the quarantine list when the total size of quarantined objects is larger than the user-defined setting.

iReplayer checks for use-after-free errors before any object is actually freed, as well as at epoch boundaries. Similar to buffer overflows, an overwritten canary indicates that a use-after-free error has occurred. iReplayer employs re-execution to identify the root cause of the error, by installing a watchpoint at the overwritten canary.

4.3 Interactive Debugging Tool

iReplayer is easy to integrate with the GDB debugger in case of abnormal exits, such as assertion failures, segmentation faults, or aborts. iReplayer intercepts these exits and stops inside the signal handler. Therefore, it is possible for programmers to find the call stack associated with abnormal exits, when the process is attached to the debugger. Inside the debugger, programmers may find the variables with a fault.

Then, it is possible for programmers to set watchpoints or breakpoints in order to identify the root causes of the fault. Afterward, the programmer can issue the rollback command from within the debugging application, which is supported by iReplayer. For instance, if watchpoints have been set, the GDB debugger will receive notifications when the corresponding addresses have been accessed. Thus, programmers are able to identify the root causes of the fault, without restarting the application.

5 Evaluation

5.1 Experimental Setup

We performed all experiments on a 16-core quiescent machine. This machine has two sockets, installed with Intel(R) Xeon(R) CPU E5-2640 processors and 256GB of memory. It has 256KB L1, 2MB L2, and 20MB L3 cache, separately. The operating system is the vanilla Linux-4.4.25. All applications were compiled using Clang-3.8.1 at the -O2 optimization level.

Evaluatated Applications: iReplayer is evaluated on a popular benchmark suite, PARSEC [13], and several widely-used real applications such as memcached-1.4.25, pbzip2, aget, pfscan, Apache httpd-2.4.25, and SQLite-3.12.0. For PARSEC applications, the native input datasets were used. pbzip2 compresses a 150MB file and pfscan scans a 826MB file. aget downloads data (614MB in total) from a machine on the same local area network, to avoid interference caused by the Internet. Memcached is evaluated using a Python script [1]. A program, called “threadest3.c”, is used to evaluate SQLite [66]. Apache is evaluated by sending 10,000 requests via the ab benchmark [2].

5.2 Identical Re-execution

We validate the identical execution by checking the order of synchronizations and system calls, as well as the final state of the heap memory. Identical re-executions should always lead to an identical heap image. The probability of a non-identical execution concluding with the same memory state is extremely low, if not completely impossible.

To perform the validation, we manually implanted a buffer overflow error in the end of the main routine for every program. This buffer overflow immediately triggers a re-execution, and we record the memory state before and after the replay. For the default library, we execute applications twice, and collect the memory differences between these two executions.

We have evaluated the identical re-execution on 15 applications, including 6 real applications, such as Apache, SQLite, and Memcached. Both iReplayer and RR can identically reproduce all of these applications, with an identical final heap image at the end. The results of the memory differences are listed in Table 1. Note that canneal cannot replay identically, since it invokes multiple atomic functions to swap two encapsulated pointers in the original program. As acknowledged in Section 1, iReplayer cannot support identical replay for applications with ad hoc synchronizations, without additional instrumentation. Therefore, we manually replaced all atomic instructions (reads/writes) with mutex locks. After this change, iReplayer achieves the same heap image, as expected. Note that it is much easier for RR to guarantee identical execution, since utilizing a single thread to run multiple threads will greatly eliminate concurrency, as discussed further in Section 6.

5.2.1 Handling Race Conditions

In our experiments, iReplayer identically reproduced 14 out of 15 applications in their first re-execution, although these applications have more than 146 race conditions in total: bodytrack(10), x264(72), streamcluster(24), ferret(38), and pbzip2(2) [25]. That is, we did not observe any divergence of the schedule for these 14 applications during their first replay. However, we are unsure as to whether these races are actually exercised, which is not our focus. Only bodytrack requires a second re-execution because of a confirmed race condition related to conditional variables [25]. Currently, iReplayer does not record the order of conditional signal and broadcast, which causes this replay issue. During
We compared the performance overhead of iReplayer with rr-4.5.0 and CLAP [35], for the recording phase. iReplayer does not support replay for the whole execution, which is why we did not evaluate performance for the replay phase. 

RR is the only available system supporting identical replay [52]. CLAP is another available RnR system for C/C++ programs that only uses software-based approaches [35]. We cannot find the source code for more recent work, such as Castor [50] and H3 [36]. Also, these two recent works actually utilize Intel’s Processor Tracing hardware feature, which only appeared after 2013 [62]. CLAP records thread-local execution paths at runtime, then computes memory dependencies offline. However, their recording mechanism is not available, even after contacting the authors. We re-implemented the recording routine of CLAP based on the path profiling support in LLVM-3.3 [35]. Paths were selected by the Ball-Larus algorithm [8], and a function call was inserted at the entrance/exit of each function, as well as back edges, in order to record the path number and function call number. Events are recorded in per-thread lists, similar to the design of CLAP. We have confirmed that the performance of aget, pfscan, and bbuf, with our implemented version, has similar performance to that of the paper [35]. We also confirmed the correctness of our implementation with the CLAP authors.

Results are listed in Table 3, where all results are normalized to the runtime with the default pthreads library. RR is compiled with Clang-3.8.1, since it cannot be compiled using the older version of Clang. Applications using CLAP and iReplayer are compiled using Clang-3.3 for fair comparison, since the implementation of the Ball-Larus algorithm is not available after Clang-3.3. CLAP cannot run four applications due to analysis errors in LLVM’s path profiling support, and RR cannot run on Apache.

On average, CLAP runs around 2.4× slower, while iReplayer only imposes negligible performance overhead (around 3%). RR runs around 17× slower, due to using a single thread to run multithreaded programs. In order to identify why iReplayer behaves better than the default glibc library, we also evaluate the performance of iReplayer’s allocator (“IR-Alloc” in Table 3. We observed that iReplayer’s custom memory allocator contributes a performance boost of about 2.5%, because it avoids the lock acquisitions of memory allocations/deallocations, since each thread is not sharing the heap with others. Also, iReplayer’s allocator avoids the large number of madvise system calls (e.g., dedup), and eliminates the possible false sharing effect [45]. Thus, the actual recording overhead of iReplayer should be around 6%.

For all applications, except fluidanimate and streamcluster, iReplayer introduces less than 10% recording overhead. Based on our investigation, which is omitted due to space limitations, fluidanimate performs over 54 million lock acquisitions every second, where recording every acquisition, and performing the synchronized checking prior to each, introduces around 49% overhead. The overhead of streamcluster mostly comes from iReplayer’s custom memory allocator, for which we do not know the exact reason.

In contrast, CLAP performs poorly in CPU-intensive applications that have a large amount of back-edges and branches, such as ferret, streamcluster, swaptions, and x264. For applications that are I/O-intensive (like aget), or applications for which most of their workload is performed in uninstrumented libraries (like pbzip2), CLAP performs very well. RR is typically very slow, except for I/O-bound applications such as aget, Memcached, and SQLite.
We further compare the performance overhead of iReplayer and its detection tools with AddressSanitizer [65], the previous state-of-the-art in detecting both buffer overflows and use-after-free errors. AddressSanitizer instruments memory accesses during compile time, and checks for possible memory errors by handling instrumented accesses. We disabled the detection of memory leaks and instrumentation of memory reads for stack and global variables in AddressSanitizer for a fair comparison. We used Clang-3.8.1 for the evaluation, as it ships with the recent version of AddressSanitizer.

As seen in Figure 5, iReplayer’s detectors (“iReplayer (OF+DP)”) only impose around 5% performance overhead on average, which is significantly lower than that of AddressSanitizer (26%). It is worth noting that AddressSanitizer cannot detect memory errors caused by non-instrumented components, which includes all libraries that these applications may invoke. This explains why AddressSanitizer has comparatively much better performance on applications that invoke many non-instrumented libraries, or perform extensive network communications, such as aget, Apache, and Memcached.

5.5 Debugging Tools
We have performed experiments on Memcached, Crasher, and all evaluated PARSEC applications using implanted buffer overflows. Crasher contains a segmentation fault, while the others have buffer overflows. All of these bugs can be caught using the interactive debugging method, as described in Section 4.3.

6 Limitations and Future Work
This section shows some limitations of iReplayer, and possible extensions in the future.

Firstly, iReplayer supports epoch-based record-and-replay, but not re-execution of the entire program. Re-executing the whole program has better diagnostic capabilities, such as identifying root causes far from the failure site. However, it has some issues as listed in Section 1. There is a chance that replaying the last epoch may miss root cause for some bugs, although existing studies show that most bugs have a very short distance of error propagation and thus should be identifiable [6, 27, 61].

Secondly, iReplayer may not achieve identical re-execution when programs with race conditions do not lead to a divergence from the recorded sequence, since it utilizes the order of synchronizations and system calls to determine whether the re-execution is identical to the original. However, it should not be a big issue, it is very unlikely to occur or is caused by benign races that users generally do not care about. As described above, iReplayer cannot support identical replay for programs with ad hoc synchronizations. This issue can be easily solved by instrumenting the code that allows iReplayer’s runtime to record these events, as Castor proposed [50]. However, we did not implement this due to two reasons: (1) it requires program instrumentation, which will create barriers for easy deployment. (2) Existing study shows that 22-67%
of ad hoc synchronization uses result in bugs or severe performance issues [72], which should be avoided as much as possible.

Thirdly, iReplayer’s detection tools support evidence-based error detection, but cannot detect problems caused exclusively by memory reads, as they do not leave behind evidence of their occurrence. Thus, while they exhibit no false positives, they will miss read-based errors.

[[Difference between races and ad hoc synchronizations: ad hoc synchronizations we currently do not support. But we can tolerate race conditions.]]

7 Related Work

7.1 Record-and-Replay Systems

There is a significant amount of record-and-replay systems. We focus on RnR systems that support multithreaded programs with race conditions, and run on the off-the-shelf hardware.

Some RnR systems require changes to the OS, such as ReVirt [24], Triage [69], Respec [41], and DoublePlay [70], which prevents widespread adoption due to security or reliability concerns related to altering the OS. Some utilize static analysis to reduce runtime overhead, such as ODR [5], LEAP [34], CLAP [35], Light [44], and H3 [36]. However, they may exhibit a scalability issue for their offline analysis.

There are also other approaches that do not require changes to the OS or offline analysis [12, 22, 28, 40, 43, 50, 53, 60, 71], which is closer to iReplayer. However, they also have their own shortcomings. Some impose more than 10x performance overhead [12, 60], R2 requires significant manual annotations to specify which functions should be monitored [28], and some require recomputation to annotate weak-locks on the race code [40].

Some existing work also prevents the recording of race accesses, which is similar to iReplayer. Arnold detects the divergence of executions caused by race conditions, and can attach a vector-lock data race detector in replay [22]. However, it relies on manual instrumentation to prevent them. Lee et al. employs multiple tries (based on a single-threaded re-execution) to search for a matched schedule [43]. iReplayer employs multiple threads to replay, and can find a matched schedule in fewer tries. Castor utilizes the hardware synchronized timestamp counters to order events, and hardware transactional memory to reduce locking overhead inside critical sections [50]. However, Castor requires instrumentation that prevents its easy deployment, and cannot replay programs identically since it does not handle the heap allocations. iReplayer does not rely on any hardware features, but employs a novel data structure and level of indirection to avoid significant recording overhead on synchronizations. Overall, iReplayer achieves a similar level of performance overhead as Castor, but can identically reproduce programs without ad hoc synchronizations.

Difference between iReplayer and RR: No existing work can guarantee identical re-execution in the in-situ setting. RR is the only available system that supports identical re-execution [52]. However, RR executes and replays multiple threads using a time-sharing method on a single core, which makes it easier to achieve identical replay. However, it is not portable to multicore hardware, which is the reason why RR runs more than 17x slower than the glibc library. Further, RR does not support in-situ replay. By comparison, iReplayer’s overhead is less than 3%, and supports in-situ replay.

7.2 Deterministic Multithreading

Deterministic multithreading (DMT) systems is another interesting direction that is distinct from this work [7, 9, 20, 21, 23, 46]. DMT systems are generally unsuitable for debugging purposes, as they can only exercise one possible schedule. Although they completely avoid recording overhead by always enforcing a deterministic order on synchronizations, they may impose much larger performance overhead when handling race conditions [20].

7.3 Detecting Memory Errors

Many dynamic approaches can detect memory errors, since they do not generate false positives. Typically, they utilize either dynamic or static instrumentation to instrument every memory access. Dynamic instrumentation tools [17, 30, 37, 55, 56] do not require the recompilation of source code, but may increase performance overhead as high as an order of magnitude. Static instrumentation tools [4, 26, 29, 54, 58, 65] may employ static analysis to help reduce the volume of instrumentation. AddressSanitizer is the state-of-the-art in dynamic analysis tools [65], which can also detect out-of-bound reads on stack and global variables, which is not available with iReplayer. However, AddressSanitizer requires instrumentation, and cannot be utilized in a production environment. DoubleTake provides similar functionality to iReplayer [47]. However, it cannot support multithreaded programs, which is very challenging to achieve, as discussed in Section 2. Also, DoubleTake does not support the same interactive debugging as that of iReplayer.

8 Conclusion

This paper introduced iReplayer, a novel system that supports identical replay in the in-situ setting. iReplayer imposes negligible performance overhead, around 3% for the recording phase, and can identically reproduce all applications without ad hoc synchronizations. To demonstrate its usefulness, three tools are built on top of it: two automatic tools for detecting the root causes of buffer overflows and use-after-free errors, and an interactive debugging tool that helps identify the source of segmentation faults and other abnormal exits. iReplayer will be publicly available upon acceptance.
References


iReplayer: In-situ and Identical Record-and-Replay for Multithreaded Applications

PL’17, January 01–03, 2017, New York, NY, USA