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

Hongyu Liu
University of Texas at San Antonio
United States
liuhyscc@gmail.com

Sam Silvestro
University of Texas at San Antonio
United States
Sam.Silvestro@utsa.edu

Wei Wang
University of Texas at San Antonio
United States
Wei.Wang@utsa.edu

Chen Tian
Huawei US R&D
United States
Chen.Tian@huawei.com

Tongping Liu
University of Texas at San Antonio
United States
Tongping.Liu@utsa.edu

Abstract

Reproducing executions 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 targets the in-situ setting, in which replay occurs within the same process as the recording process. Also, most existing work cannot achieve identical replay, which may prevent the reproduction of some errors.

This paper presents `iReplayer`, which aims to **i**dentically **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. Currently, `iReplayer` only incurs 3% performance overhead on average, 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`.

CCS Concepts • **Computer systems organization → Reliability**; • **Software and its engineering → Software testing and debugging**;

Keywords Record-and-Replay, Identical Replay, In-situ Replay, Multithreaded Debugging

PLDI'18, June 18–22, 2018, Philadelphia, PA, USA

ACM ISBN 978-1-4503-5698-5/18/06…$15.00
https://doi.org/10.1145/3192366.3192380

ACM Reference Format:
Hongyu Liu, Sam Silvestro, Wei Wang, Chen Tian, and Tongping Liu. 2018. `iReplayer`: In-situ and Identical Record-and-Replay for Multithreaded Applications. In *Proceedings of 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18).* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3192366.3192380

1 Introduction

Multithreaded programs contain intrinsic non-deterministic factors that may affect the schedule and results of different executions. Thus, reproducing multithreaded programs is very challenging. Record-and-Replay (RnR) systems record non-deterministic events of the original execution, such as the order of synchronizations and the results of certain system calls, and then reproduce these events during the re-execution [62]. Some RnR systems even record the order of memory accesses [13], or utilize offline analysis to infer the order of memory accesses inside the execution [5, 37, 38, 46]. However, existing RnR systems have two shared shortcomings, in addition to their specific problems as described in Section 7.

First, they do not support in-situ replay, typically reproducing the execution in a different process. They could possibly achieve better diagnostic capability, since they can access all information from the entire execution [6]. However, there are several issues. (1) As observed by experts [40, 71], offline replay requires the same runtime environment as the recording process, which will greatly limit their usage, since normal users may not want to share third-party libraries or sensitive inputs/logs with programmers due to business and privacy concerns. (2) They cannot be utilized to assist online recovery [71].

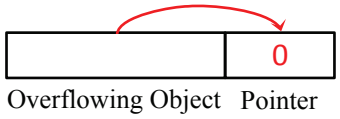

Figure 1. A null reference problem.

Second, most existing RnR systems (except RR [55, 60]) cannot identically reproduce the recorded execution, as they do not guarantee the same system states, such as process/thread IDs and file descriptors [5, 16, 33–35, 37, 46, 52, 54, 62, 66, 72], the same results of system calls (e.g., `time`) [36, 37, 48], or have different memory layouts [5, 16, 33–35, 37, 46, 52, 54, 62, 66, 72]. Therefore, it is impossible to reproduce some types of bugs: (1) Bugs related to memory layout may not be reliably reproduced. Figure 1 shows such an example, in which a crash occurs when dereferencing a null pointer caused by a buffer overflow. 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 dependent on system states, such as thread IDs, file descriptors, or memory addresses, may not be reproducible, when the states in replay are not the same as those in the original execution. However, many real systems were designed to utilize 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 [41].

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.

First, iReplayer designs an in-situ replay technique that always replays the last-epoch execution within the same process as the original execution. The in-situ replay makes it easier to replay identical system states and is more likely to reproduce bugs. This in-situ replay is different from existing online replay [44, 64, 71, 72], where their replays 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, 29, 64], most bugs have a very short distance of error propagation, which indicates a root cause may be located shortly prior to failures. Replaying-last-epoch also avoids significant time spent waiting for problems to appear.

Second, 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 (even racy ones) that do not contain implicit synchronizations (i. e., without using pthread APIs).

Third, iReplayer only imposes 3% recording overhead on average, which is sufficiently low for deployment. `iReplayer` utilizes multiple approaches to reduce its logging overhead: (1) it takes advantage of the in-situ setting to avoid recording the content of file reads/writes; (2) It avoids the recording of memory accesses by handling race conditions in replay phases, inspired by existing work [45, 52]; (3) It avoids the recording of memory allocations by employing a novel heap design, inspired by Dthreads [48]; (4) It also designs a novel data structure to efficiently record the local-order of synchronizations, while still ensuring identical replay; (5) `iReplayer` designs an indirect level for recording synchronization events, similar to existing work [4]. These are the major reasons why `iReplayer` has much less overhead than past related techniques–e.g. Respec [44]. More details can be seen in Section 3.2.

The identical and in-situ re-execution of iReplayer enables a range of possibilities, and three tools are shown in this paper. These tools can be utilized in staging or canary deployment, especially when new features are rolling out. In addition, the in-situ and identical replay of `iReplayer` enables unique possibilities: (1) It enables on-site tools that can automatically diagnose root causes of program failures, such as memory errors, segmentation faults, aborts, and assertions. For instance, upon faults, we could perform binary analysis to pinpoint faulting addresses, then install watchpoints on them to identify root causes on-site without human involvement. In contrast, offline RnR cannot perform on-site analysis, and typically require additional human effort. (2) It enables evidence-based approaches to prevent program failures, such as memory errors or deadlocks. For instance, it is possible to extend `iReplayer` to delay memory deallocations to prevent discovered use-after-frees, or enforce an alternative lock order to avoid deadlocks. It is impossible to perform online repair with existing offline RnRs.

Overall, this paper makes the following contributions:

First in-situ record-and-replay technique for multithreaded programs: `iReplayer` proposes the first in-situ RnR system that the replay occurs in the same process as the original execution, enabling new possibilities.

An identical replay technique: `iReplayer` supports the identical replay of multithreaded programs without self-defined synchronizations. The identical replay helps reproduce bugs, and ease the development of automatic tools.

Practical implementation techniques to reduce overhead: `iReplayer` makes multiple design choices to reduce recording overhead: it proposes a novel data structure that supports identical replay with low recording overhead, and supports the checking of divergence easily during the replay; it designs a novel heap to avoid the recording of memory allocations.

A practical system combining low recording overhead 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 nonexistent hardware, customized OS, or the modification of programs.

Multiple promising applications: To demonstrate the usefulness of iReplayer, this paper developed two tools to detect heap over-writes and use-after-free errors, and one interactive debugging tool (connecting with GDB).

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 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), abnormal exits, or user-defined criteria. For instance, users may use the size of logging as the criteria, in order to reduce memory/disk consumption.

The overview of iReplayer is illustrated in Figure 2, which shows an execution with two threads. At the beginning of an epoch, iReplayer takes a snapshot of the program's states, such as its memory and the position of open files, so that the program can be rolled back to this point (Section 3.1). During the original execution, iReplayer records the order of synchronizations (Section 3.2), and handles system calls differently (Section 2.2.3). When a thread encounters an irrevocable system call – which changes the state, but cannot be safely rolled back – or reaches the user-defined criteria for recording, it will be treated as the coordinator thread, and will coordinate with other threads to pause the execution (Section 3.3). After all threads have reached a quiescent state, the coordinator thread determines whether to continue the execution, or perform re-execution, based on user instructions or tool-specific evidence (see Section 4). If a replay is required, the coordinator notifies all other threads to roll back (Section 3.4) and re-execute the program (Section 3.5). Otherwise, all other threads are notified to proceed to the next epoch.

2.2 Challenges for In-situ and Identical Replay

The major challenge of iReplayer is to ensure identical replay of synchronizations, memory accesses, system calls, and memory layout under the in-situ setting.

2.2.1 Ensuring Identical Synchronizations

We record the order of synchronizations during the original execution, then replay this order in re-executions [62]. However, the greatest challenge is to achieve efficient recording, which is further described in Section 3.2. Since some applications rely on the results of synchronization functions, such as try locks or barrier waits, iReplayer also records the return values of synchronizations and returns them in replays.

Currently, iReplayer does not support programs with ad hoc synchronizations, where programs use their own synchronization methods rather than explicit pthreads APIs [74]), as further discussed in Section 6.

2.2.2 Ensuring Identical Order of Memory Accesses

Two types of memory accesses exist in multithreaded applications, including thread-private and shared accesses. The order of thread-private accesses is determined by the order of instructions, which does not require special handling for identical replay. Shared accesses will be identical if they are properly protected by explicit synchronizations, when explicit synchronizations are identically reproduced. Thus, the difficulty lies in ensuring the identical replay for race conditions.

Handling race conditions: iReplayer does not record racy accesses initially, since that is too expensive [13]. Instead, it handles race conditions inside replay phases, which avoids significant recording overhead for common cases in which programs do not expose race conditions. During replay, it will check for the divergence from the recorded events. If a replay behaves exactly the same as the original execution, i.e. the same order of system calls and synchronizations, then race conditions are either not exposed or are successfully reproduced. Otherwise, iReplayer immediately initiates another replay, and utilizes multiple replays to search for a matched schedule. When the events of a replay match the recorded ones, iReplayer assumes that the replay is identical to the original execution, and will stop searching.

2.2.3 Ensuring Identical System Calls

Existing RnR systems record the results of system calls, and replay the same states during replay [62, 67]. For instance, the recorded results of gettimeofday will be returned in the replay phase. However, no existing work aims for the in-situ setting. The in-situ setting imposes some additional challenges toward ensuring identical system calls. For instance, if some sequences of file-related system calls including open and close are later replayed, it may be impossible to ensure

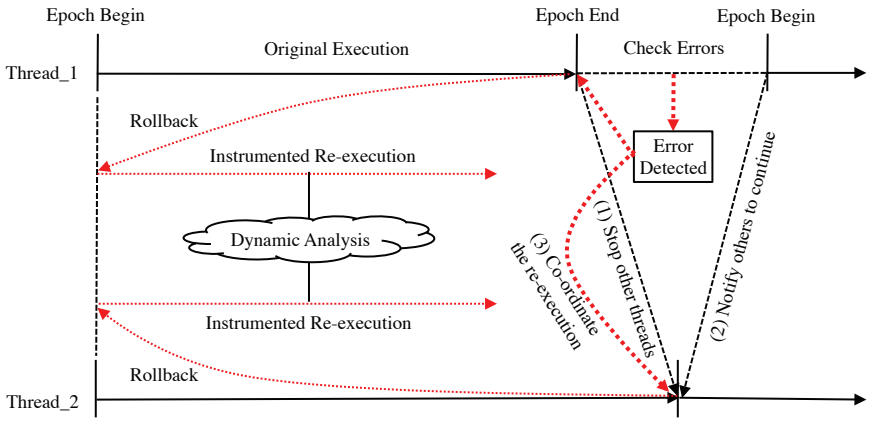

Figure 2. Overview of `iReplayer` with two threads. Bold lines represent normal program executions, and dashed lines illustrate the assistance functions at epoch boundaries. Dash-dot lines and red short-dash lines are tool-specific functions.

the same file descriptor for `open`, since it may now be occupied. Similar results may occur for the `munmap` system call. `iReplayer` classifies system calls into five categories, similar to DoubleTake [49].

Repeatable system calls always return the same results within the in-situ setting, e.g. `getpid()`. They require no special handling in either the recording or replaying phases.

Recordable system calls return different results when invoked during re-executions, such as `gettimeofday()` and socket reads/writes. `iReplayer` records the results, and returns the same values during replay without actual invocations.

Revocable system calls modify system states, but the results of these operations can be reproduced identically under the in-situ setting, as long as initial states are recovered before the re-execution. These system calls mainly include file-related reads/writes. Although their results can be recorded, this may impose substantial recording overhead [44]. Instead, `iReplayer` records the positions of open files during the recording phase, and issues these system calls normally during replays (after recovering positions).

Deferrable system calls irrevocably change system states, but can be safely delayed. These system calls, such as `munmap` and `close`, are very important for identical re-execution in an in-situ setting. `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.

Irrevocable system calls irrevocably change system states, and cannot be rolled back safely or deferred easily. Although conceptually they can be recorded as in other existing RnR systems [5, 52, 62], this may involve substantial performance overhead or engineering effort. Currently, `iReplayer` simply treats them as irrevocable system calls, and closes the current epoch when encountering them. For instance, `execve` and `fork` are examples of such 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, checkpointing, and cleaning upon epochs. Thus, irrevocable systems 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 management is a major source of non-determinism in multithreaded applications. First, the OS may randomize memory uses due to the ASLR mechanism [17]. Second, multiple threads may compete with each other. To ensure the identical memory layout, `iReplayer` isolates its internal memory uses from those of applications, adapts a "per-thread heap" so that memory allocations inside the same heap completely depend on the program order, and controls interactions among different threads.

Per-thread Heap: Built on top of HeapLayers [11], it adapts the per-thread heap organization of Hoard [10]. Memory allocations and deallocations within each thread will be identically reproduced, if they do not interfere with other threads. Different from Hoard, two live threads are never allocated from the same per-thread heap. `iReplayer` intercepts thread creation, and deterministically assigns a unique heap for every thread by utilizing a global lock to serialize thread creation. When the order of locks is replayed deterministically, each thread will have the same heap during re-executions.

Deterministically fetches blocks for per-thread heaps: `iReplayer` maintains a super heap that holds a large number of blocks for all per-thread heaps. When a per-thread

heap exhausts its memory, it obtains a new block from the super heap under the protection of a global lock, which is guaranteed to be the same during re-executions via deterministically replaying lock acquisitions.

Handles deallocations by a different thread deterministically: iReplayer always returns a freed object to the current thread issuing the free, no matter which thread allocated the object initially. Thus, this freed object only affects the subsequent memory allocations of the current thread, which again depends on the program order, and will be deterministic.

Inside each per-thread heap, objects are managed using power-of-two size classes. During allocations, each request will be aligned to the next power-of-two size. The free list will be checked first, and only if the request cannot be allocated from its free list, it will be allocated using the bump pointer mechanism [12]. Upon deallocation, each deallocated object will be inserted into the head of its corresponding free list and will be reutilized consequently. Since iReplayer limits memory allocations to its per-thread heap and controls the interactions among different threads, there is no need to record the addresses of allocations to ensure identical replay. Note that iReplayer does not serialize memory allocations, but only the acquisition of each block (4 megabytes). Instead, iReplayer avoids the usage of locks upon each allocation, which explains why its heap is 3% faster than the default Linux allocator (Section 5.3).

2.3 Other Challenges

There are other challenges, mostly caused by the in-situ setting: how to perform recording efficiently (Section 3.2)? How to stop an epoch safely under the in-situ setting (Section 3.3), when some threads may be in the middle of a system call or waiting for synchronizations? How to roll back multiple threads correctly, especially for threads created in the last epoch or are waiting on synchronizations (Section 3.4)? How to prepare for re-execution (Section 3.4) to assist identical replay? How to control the order of re-executions, and detect divergence possibly caused by race conditions (Section 3.5)?

3 Implementation

This section describes the implementation of iReplayer, organized by phases that are shown as Figure 2.

The start of a program is considered the start of the first epoch, and terminations (either normal or abnormal exits) will be treated 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 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-executions.

3.1 Epoch Begin

At epoch begin, the major task is to checkpoint the states of the execution in order to support re-executions. 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 unnecessary 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, which will be issued at this time. Cached data for closed sockets will be removed, and joined threads will be reclaimed.

After this, iReplayer checkpoints the states shared by all threads, such as 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. iReplayer also updates file positions of all open files, which are tracked in a global hash table.

Afterwards, all other threads are woken up, including threads waiting on condition variables, barriers, and thread joins, so that they can checkpoint 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. 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. Each event is recorded in its per-thread list initially, then will be added into the corresponding per-variable list. For the example shown in Figure 3, the corresponding order will be recorded as in Figure 4. For instance, *lock*1's per-variable list will track that *lock*1 is first acquired by Thread1, and then by Thread2.

This data structure removes the need for the global order [62], reconstructing the synchronization order offline [37, 38, 52], and special hardware support [52]. It guarantees identical re-execution of explicit synchronizations: different synchronizations inside the same thread will always have the same order, determined by its program logic; the per-variable

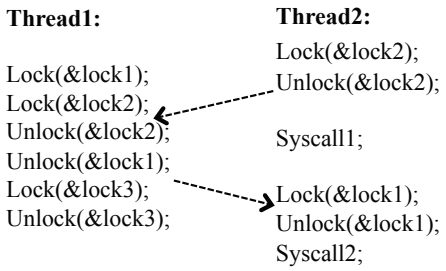

Figure 3. Code snippet with sync and syscalls.

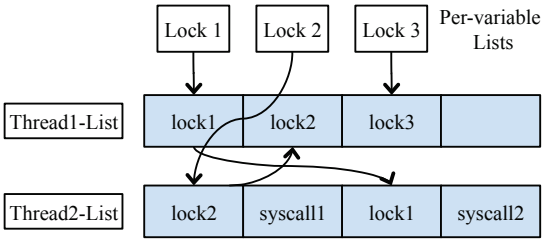

Figure 4. Data structures for tracking events.

list ensures that multiple threads will perform synchronizations in the recorded order. In addition, this data structure is convenient for checking the divergence during re-executions: each thread is only required to check whether the next event is the same as the recorded one in their per-thread lists. If not, this is an indication of a divergence, and `iReplayer` will immediately invoke a re-execution in order to search for a matching schedule.

This data structure is very efficient at tracking events: (1) Its recording does not introduce new lock contention between threads, excepting the original lock operations, which is different from existing work [44]. (2) `iReplayer` further reduces its logging overhead by pre-allocating a specified number of entries for per-thread lists, which does not require additional memory allocations during the recording. When all entries are exhausted, it is time to stop the current epoch and start a new epoch.

Second, `iReplayer` employs a level of indirection for finding the list associated with each synchronization variable, instead of naively using a global hash table, similar to SyncPerf [4]. It is difficult to define the size of the hash table and design a balanced hash algorithm. The naive method was found to impose up to 4× performance overhead when applications has a large number of synchronization variables, e.g. `fluidanimate` of PARSEC [14]. Instead, `iReplayer` allocates a shadow synchronization object from its internal heap (to avoid interfering with the application's memory uses), and saves the pointer to this shadow object within the first word of the original synchronization object. This shadow object includes the real synchronization object and a pointer to its per-variable list.

Third, `iReplayer` avoids the recording of each memory access by delaying the handling of race conditions until the replay phase (Section 2.2.2), avoids the recording of memory allocations by employing a novel heap (Section 2.2.4), and avoids the recording of file reads/writes by treating them as revocable system calls (Section 2.2.3).

3.2.1 Supporting Synchronizations

`iReplayer` supports a range of synchronization primitives, such as thread creations, various forms of mutex locks, condition variables, barriers, signals, and thread joins.

Thread creation, destruction, and joins: `iReplayer` intercepts `pthread_create` function calls to initialize thread-related data, checkpoint the state prior to the execution, and handle future thread exits. It takes multiple approaches to guarantee identical replay: (1) It does not allow concurrent thread creations by using a global mutex; (2) `iReplayer` keeps threads alive (without exiting) until the next epoch in order to preserve system states, such as thread IDs and stacks, by using a thread-specific condition variable and a status field. For a joinee thread 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 joinee thread wakes its joiner immediately. Joinee threads are always waiting on a condition variable (and thus kept alive), awaiting notification to 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.

Condition 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 condition variables, similar to lock acquisitions. Since other threads may close the current epoch during waiting, every thread should record its state and which condition variable before 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 `cond_signal` and `cond_broadcast`, but only the wake-up order of threads. Note that this method may induce a non-identical replay when locks are not properly acquired. `iReplayer` overcomes this by inserting random sleeps at diverging points, as shown in Section 5.2.

Barriers: During barrier waiting, a thread may wait inside the kernel until being joined by the required number of threads. However, it is difficult to wake up a thread waiting on the barrier. To solve this issue, `iReplayer` re-implements the barrier by combining a mutex and a condition variable. `iReplayer` intercepts the initialization of barriers in order

to initialize the corresponding mutex locks and condition variables. iReplayer does not record the order of entries into a barrier, since a thread waiting on a barrier will not change the state. Instead, iReplayer records the return value of every barrier wait, since some applications may rely on it.

3.3 Epoch End

At epoch end, the coordinator thread is responsible for 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 adapts the "stop the world" approach employed by garbage collection [15]. Stopping an epoch safely is unique to the in-situ setting, which has several challenges:

Challenge 1: How to stop other threads safely? Asynchronous methods (e.g. signals) are unreliable to stop and roll back threads cleanly, if these threads are waiting inside the kernel due to synchronizations (such as barrier_wait) or other system calls. Instead, iReplayer employs a synchronized method: before any synchronization or system call invocation, it checks whether a coordinator thread has requested to stop the current epoch. If so, the current thread will wait on its internal condition variable, and mark its state as stopped.

Challenge 2: How to stop threads waiting on synchronizations or blocking on external system calls? Threads waiting on condition 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. 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. Some blocking IOs will be turned into non-blocking IOs upon interceptions, e.g. adding the timeout value for epoll_wait.

When all other threads (except the coordinator thread) have been stopped, the current epoch is closed. Thus, the coordinator thread will 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 orchestrates these operations.

3.4 Preparing for Re-execution

iReplayer prepares the re-execution in the following steps.

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

Secondly, iReplayer resets pointers of all per-thread lists and per-variable lists to their first recorded entry. Internally, iReplayer maintains a hash table to track the mapping between synchronization variables and their shadow objects to assist this.

Thirdly, iReplayer recovers file positions of all opened 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 condition variables or barriers should first be woken up. However, threads created during the last epoch should wait for notifications from their parents, after their corresponding thread-creation events have transpired.
(2) Rolling back the stack should be performed very cautiously, since the stack to be recovered might overwrite live values on the current stack, which can cause a program to behave abnormally. iReplayer forces all threads to use temporary stacks before copying, then switch back to their original stack after the copy has completed. (3) Because iReplayer only restores the used portion of the stack in the last checkpoint, the remainder of the stack should be zeroed out to guarantee identical replay. Some applications contain un-initialized reads that may access stack variables beyond the stack of the last checkpoint. (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. For these cases, iReplayer passes control to a custom function by setting the IP pointer, so that the rollback can be performed in this function after returning from 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-executions, 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. Note that iReplayer's replay is different from Castor [52]. Castor requires the construction of the order of synchronizations by using timestamps of different synchronizations. iReplayer designs a novel data structure (Section 3.2) to overcome this issue, which makes it suitable for in-situ setting.

For each thread, iReplayer utilizes a condition variable and a mutex lock to control the re-execution, and relies on per-thread lists and per-variable lists to guide the re-execution (as described in Section 3.2). **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 on its condition variable until previous synchronizations on it have transpired.

iReplayer utilizes a global mutex to control the order of thread creations. During re-executions, the parent thread waits for its turn to proceed, then notifies the corresponding child thread (waiting on their internal condition 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 the 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 they are expected by comparing with the recorded events. For synchronizations, iReplayer confirms whether the address and type of synchronization is expected. Any divergence from the recorded ones can be only caused by unknown race conditions, when all explicit synchronizations and system calls are replayed faithfully. If a divergence is detected, iReplayer will restart a new execution immediately. It will stop performing re-executions when a run with the same events as those recorded is found. Currently, iReplayer supports an unlimited number of replays. Note that since iReplayer replays all explicit synchronizations and system calls, it generally takes very few re-executions to find a matching schedule, as evaluated in Table 2. If the replay cannot reproduce the original schedule, iReplayer may insert random delays at diverging points, but without changing the order of the recorded schedule. Therefore, this mechanism helps reproduce applications with race conditions, as shown in Section 5.2.

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 [49]. These applications exemplify the usefulness of an in-situ and identical record-and-replay system as iReplayer.

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 [20]. 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 by invoking the perf_event_open system call. During re-execution, instructions writing to the watched addresses will trigger a trap, such that iReplayer reports the complete call stack of the faulted instruction that causes an overflow. Since there are four watchpoints, iReplayer can identify root causes of four buffer overflows in one re-execution simultaneously. 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 per-thread quarantine lists, an idea originally developed by AddressSanitizer [68]. iReplayer fills the first 128 bytes of freed objects with canary values. These freed objects are released from the quarantine list when their total size becomes larger than the user-defined setting.

iReplayer checks for use-after-free errors before any object is actually removed from the quarantine lists, 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-executions to identify the root cause of each error, by installing watchpoints at overwritten canaries. During re-execution, iReplayer stores the call stack of allocations and deallocations for the purpose of reporting. It can precisely pinpoint the statements at use-after-free sites by using the watchpoint mechanism.

4.3 Interactive Debugging Tool

iReplayer designs an interactive debugging tool 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 addresses of faulted variables, and set watchpoints on these addresses. Afterward, the programmer can issue the rollback command via the debugger, 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 buggy application. This interactive debugging tool will not only help programmers identify faults in development phases, but can also be utilized in staging deployment [53], especially when new features are rolling out.

5 Evaluation

5.1 Experimental Setup

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

Evaluated Applications: iReplayer was evaluated using PARSEC 2.1 [14] and several 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 compressed a 150MB file and pfscan scanned a 826MB file. aget downloaded 614MB of data from a machine on the same local area network, to avoid interference caused by the Internet. Memcached was evaluated using a Python script [1]. A program, called "threadtest3.c", was used to evaluate SQlite [69]. Apache was evaluated by sending 10,000 requests via the ab benchmark [2].

5.2 Identical Re-execution

We validated 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. Currently, we did not evaluate explicit outputs, such as file or socket writes, although these evaluations could increase the confidence of the identical execution.

To perform the validation, we manually implanted a buffer overflow error in the end of 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 Linux library, we collected the memory differences between these two executions. We also evaluated the memory differences of RR, the only work supporting identical re-execution without special OS support [55].

We evaluated the identical re-execution on 15 applications, including 6 real applications. The results of memory differences are listed in Table 1. Note that canneal cannot be replayed identically initially, since it invokes multiple atomic functions to swap two encapsulated pointers in the original program. As discussed in Section 1, iReplayer cannot support identical replay for applications with ad hoc synchronizations, if without additional instrumentation. Therefore, we manually replaced all atomic instructions (reads/writes) with mutex locks. After these changes, iReplayer achieves the same heap image, as expected. Therefore, both iReplayer and RR can identically reproduce all of these applications, with an identical final heap image at the end.

5.2.1 Handling Race Conditions

In our experiments, iReplayer identically reproduced 14 out of 15 applications (including modified canneal) 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) [27]. That is, we did not observe any divergence of the schedule for these 14 applications during their first replay. However, we are not sure regarding whether these races are actually exercised. Only bodytrack requires a second re-execution because of a confirmed race condition related to condition variables [27]. Currently, iReplayer does not record the order of condition signal and broadcast, which causes this replay issue. During the second replay, iReplayer successfully reproduces this program by inserting some delays upon diverging points, which avoids the race condition.

In order to further confirm how race conditions affect replay, we also evaluated a synthetic racy program—Crasher [51]. We ran Crasher 100,000 times, and the race condition (causing a crash) was observed on 82,592 out of 100,000 executions. Note that normal applications will not have such a high probability of exhibiting a race, since this program intentionally places sleep inside to trigger the race condition. The results of reproducing race conditions are further shown in Table 2. In around 99.87% of executions, iReplayer reproduced the race condition during the first replay. Approximately 0.11% of the time, iReplayer required two replays to reproduce the race, while the remaining ones (0.02%) required more than two replays. These results indicate that iReplayer has an excellent chance to reproduce races, when all other explicit synchronizations are replayed faithfully.

5.3 Performance Overhead

We compared the performance overhead of iReplayer with rr-4.5.0 and CLAP [37], for the recording phase. iReplayer does not support replay for the whole execution, which

Table 1. The percentage of memory difference between the original execution and the re-execution for the default library ("Orig") and iReplayer ("IR"). BS is the abbreviation for the blackscholes.

	BS	bodytrack	canneal	dedup	ferret	fluidanimate	streamcluster	swaptions	x264	aget	apache	memcached	pbzip2	pfscan	sqlite
Orig	3	9	43	25	5	7	< 0.1	8	24	7	12	4	4	5	10
IR	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
RR	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 2. Results of reproducing Crasher's race.

Replay Times	1	2	3	≥ 4
Percentage	99.8718	0.1088	0.0121	0.0073

is the reason why we did not evaluate performance for the replay phase.

RR is the only available system supporting identical replay [55]. CLAP is another available RnR system for C/C++ programs that only uses software-based approaches [37]. We cannot find the source code for more recent work, such as Castor [52] and H3 [38]. Also, these two recent works actually utilize Intel's Processor Tracing hardware feature, which only appeared after 2013 [65]. CLAP records thread-local execution paths at runtime, then computes memory dependencies offline. However, their recording mechanism is not available. We re-implemented the recording routine of CLAP based on the path profiling support in LLVM-3.3 [37]. 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 reported in the paper [37]. 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 of 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 investigate why iReplayer behaves better than the default Linux library, we also evaluated the performance of iReplayer's allocator (noted as "IR-Alloc" in Table 3). We observed that iReplayer's custom memory allocator contributes a performance boost of about 2.5%, but with worse performance on four evaluated applications. Based on our understanding, there are multiple reasons that can help boost the performance: (1) the allocator avoids the lock acquisitions of memory allocations/deallocations, since each thread is not sharing the heap with others. (2) iReplayer's allocator avoids the large number of madvise system calls (e.g., dedup), and eliminates the possible false sharing effect [47]. 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 (omitted due to the space limit), fluidanimate has over 54 million lock acquisitions per second, where recording every acquisition and performing the synchronized checking prior to each, adds 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 and Memcached. For other applications, RR is very slow as expected, since it cannot take advantage of the parallelism provided by multiple cores.

5.4 Detection Tools

We also evaluated the effectiveness and performance of detection tools built on top of iReplayer. This evaluation is conducted using applications with real and implanted bugs.

5.4.1 Detection Effectiveness

We confirmed iReplayer's effectiveness on heap overflows and use-after-free bugs that were collected from prior tools [49, 75], Bugbench [50], and Bugzilla [42]. These applications include bc-1.06, bzip2 [42], gzip-1.2.4, libHX, polymorph, Memcached [70], and libtiff [19]. iReplayer can detect all of these known problems, which is similar to DoubleTake.

As described in Section 5.2, we have manually inserted a buffer overflow error at the end of all evaluated applications. iReplayer's detector can also detect all of these implanted errors. Also, iReplayer reports the root causes of these bugs, with precise calling contexts of the faults.

Table 3. Performance overhead

Applications	IR-Alloc	iReplayer	CLAP	RR
blackscholes	1.001	1.021	1.113	8.011
bodytrack	0.993	0.990	-	27.283
canneal	0.880	0.962	-	6.884
dedup	0.664	0.817	1.074	5.138
ferret	1.017	0.998	3.519	13.275
fluidanimate	1.044	1.493	2.183	34.082
streamcluster	1.102	1.100	2.383	52.280
swaptions	0.979	0.990	2.964	29.921
x264	0.991	1.032	9.100	16.228
aget	1.000	1.032	1.013	1.065
apache	1.002	1.056	-	-
memcached	1.000	1.001	1.001	1.822
pbzip2	0.974	0.895	-	26.852
pfscan	1.015	1.006	1.032	8.462
sqlite	0.973	1.087	3.853	15.059
average	0.976	1.027	2.658	17.597

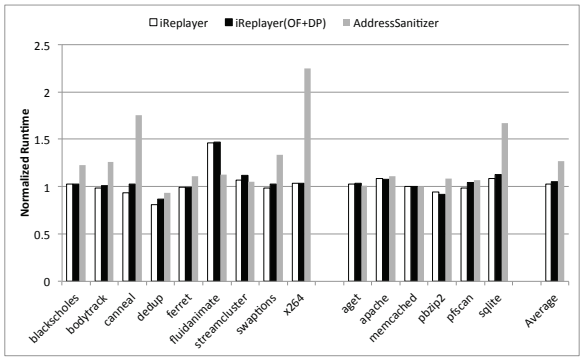

Figure 5. Comparing iReplayer's performance against AddressSanitizer in detecting memory errors.

5.4.2 Performance Overhead

We further compared the performance overhead of iReplayer and its detection tools with AddressSanitizer [68], 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. For a fair comparison, we only enable the instrumentation on memory writes on heap objects, while disabling its leak detection and others. Note that we did not instrument memory writes on all external libraries. 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 (noted as "iReplayer (OF+DP)") only impose around 5% performance overhead on average, which is significantly lower than that of AddressSanitizer (26%). iReplayer's detectors perform better than, or similar to, AddressSanitizer in almost all applications, except fluidanimate. The overhead of iReplayer on this application comes from the recording of synchronizations, as discussed above. It is worth noting that AddressSanitizer cannot detect memory errors caused by non-instrumented components, which includes all external 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 performed experiments on Memcached, Crasher [51], and evaluated PARSEC applications using implanted buffer overflows. Crasher contains a segmentation fault, while the others have buffer overflows. Each bug can be caught using the interactive debugging method, described in Section 4.3.

6 Limitations and Future Work

This section discusses some limitations of iReplayer, and possible extensions in the future.

Firstly, iReplayer only 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, 29, 64].

Secondly, it may not achieve identical re-executions 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 one. As described above, iReplayer cannot support the identical replay of programs with ad hoc synchronizations, which utilize self-defined synchronizations instead of explicit pthreads APIs. Also, these synchronizations include C/C++ atomics [21]. This issue can be solved by instrumenting the code, as Castor proposed [52], which allows iReplayer's runtime to record the order of such events. 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 [74], which should be avoided as much as possible.

Thirdly, iReplayer's detection tools support evidence-based error detection, which 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.

7 Related Work

7.1 Record-and-Replay Systems

A significant amount of record-and-replay systems exists. 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 [26], Triage [71], Respec [44], and DoublePlay [72], 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 [36], CLAP [37], Light [46], and H3 [38]. However, they may exhibit a scalability issue for their offline analysis.

Several approaches require no changes to the OS nor offline analysis [13, 24, 30, 43, 45, 52, 56, 63, 73], which is closer to `iReplayer`. However, they also have shortcomings. Some impose more than 10× performance overhead [13, 63], R2 requires significant manual annotations to specify which functions should be monitored [30], and some require recompilation to annotate weak-locks on the racy code [43].

Similar to `iReplayer`, some existing work also avoids the recording of racy accesses. Arnold detects the divergence of executions caused by race conditions, and can attach a vector-lock data race detector in replay [24]. However, it relies on manual instrumentation to fix them. Lee et al. employ multiple tries (based on a single-threaded re-execution) to search for a matching schedule [45], while `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 [52]. Thus, Castor requires hardware support and compiler instrumentation that prevents easy deployment. `iReplayer` does not rely on any hardware feature, 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 in the offline setting [55]. However, `RR` executes and replays multiple threads using a time-sharing method on a single core, which makes it easier to achieve identical replay. Due to the lack of scalability to multicore hardware, `RR` runs more than 17× slower. Further, `RR` does not support in-situ replay. By comparison, `iReplayer`'s overhead is less than 3%, and supports the in-situ replay.

7.2 Deterministic Multithreading

Deterministic multithreading (DMT) systems is another interesting direction that is distinct from this work [7, 9, 22, 23, 25, 48]. 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 [22].

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 do not require the recompilation of source code [18, 32, 39, 58, 59], but may increase performance overhead as high as an order of magnitude. Static instrumentation tools may employ static analysis to help reduce the volume of instrumentation [3, 28, 31, 57, 61, 68]. AddressSanitizer is the state-of-the-art in dynamic analysis tools [68], which can also detect out-of-bound reads on stack and global variables that is available with `iReplayer`. However, AddressSanitizer requires explicit instrumentation, and cannot be utilized in a production environment due to its prohibitive performance overhead. DoubleTake provides similar functionality to `iReplayer` [49]. 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 only approximately 3% recording overhead, 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.

Acknowledgments

This work was initiated and partially conducted while Tongping Liu was a Ph.D. student at the University of Massachusetts Amherst, under the supervision of Professor Emery Berger. We would like to thank our shepherd, Yannis Smaragdakis, and anonymous reviewers for their valuable suggestions and feedback. We thank Charlie Curtsinger, Bobby Powers, and Jinpeng Zhou for their participation in development, and Xiangyu Zhang, Michael D. Bond, Jia Rao, Corey Crosser, and Mary Mays for their helpful comments. We also thank Jeff Huang, Kostya Serebryany, Nuno Machado, Jason Flinn, Brandon Lucia, Kaushik Veeraraghavan, and David Devecsery for their help on the evaluation. This material is based upon work supported by the National Science Foundation under Award CCF-1566154 and CCF-1617390. The work is also supported by Google Faculty Award, Mozilla Research Grant, and the startup package from University of Texas at San Antonio.

References

[1] 2017. Pure python memcached client. https://pypi.python.org/pypi/python-memcached.

[2] ab Developers. 2017. ab - Apache HTTP server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html.

[3] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. 2009. Baggy bounds checking: an efficient and backwards-compatible defense against out-of-bounds errors. In *Proceedings of the 18th conference on USENIX security symposium (SSYM'09)*. USENIX Association, Berkeley, CA, USA, 51–66. http://dl.acm.org/citation.cfm?id=1855768.1855772

[4] Mohammad Mejbah ul Alam, Tongping Liu, Guangming Zeng, and Abdullah Muzahid. 2017. SyncPerf: Categorizing, Detecting, and Diagnosing Synchronization Performance Bugs. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*. ACM, New York, NY, USA, 298–313. https://doi.org/10.1145/3064176.3064186

[5] Gautam Altekar and Ion Stoica. 2009. ODR: Output-deterministic Replay for Multicore Debugging. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)*. ACM, New York, NY, USA, 193–206. https://doi.org/10.1145/1629575.1629594

[6] Joy Arulraj, Guoliang Jin, and Shan Lu. 2014. Leveraging the Short-term Memory of Hardware to Diagnose Production-run Software Failures. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. ACM, New York, NY, USA, 207–222. https://doi.org/10.1145/2541940.2541973

[7] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. 2010. Efficient System-enforced Deterministic Parallelism. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*. USENIX Association, Berkeley, CA, USA, 1–16. http://dl.acm.org/citation.cfm?id=1924943.1924957

[8] Thomas Ball and James R. Larus. 1996. Efficient Path Profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 29)*. IEEE Computer Society, Washington, DC, USA, 46–57. http://dl.acm.org/citation.cfm?id=243846.243857

[9] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. 2010. CoreDet: A Compiler and Runtime System for Deterministic Multithreaded Execution. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*. ACM, New York, NY, USA, 53–64. https://doi.org/10.1145/1736020.1736029

[10] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. 2000. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*. ACM, New York, NY, USA, 117–128. https://doi.org/10.1145/378993.379232

[11] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. 2001. Composing High-performance Memory Allocators. In *Proceedings of the ACM SIGPLAN 2001 on Programming Language Design and Implementation (PLDI '01)*. ACM, New York, NY, USA, 114–124.

[12] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. 2002. Reconsidering Custom Memory Allocation. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '02)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/582419.582421

[13] Sanjay Bhansali, Wen-Ke Chen, Stuart de Jong, Andrew Edwards, Ron Murray, Milenko Drinić, Darek Mihočka, and Joe Chau. 2006. Framework for Instruction-level Tracing and Analysis of Program Executions. In *Proceedings of the 2Nd International Conference on Virtual Execution Environments (VEE '06)*. ACM, New York, NY, USA, 154–163. https://doi.org/10.1145/1134760.1220164

[14] Christian Bienia and Kai Li. 2009. PARSEC 2.0: A New Benchmark Suite for Chip-Multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*.

[15] Hans-J. Boehm, Alan J. Demers, and Scott Shenker. 1991. Mostly Parallel Garbage Collection. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation (PLDI '91)*. ACM, New York, NY, USA, 157–164. https://doi.org/10.1145/113445.113459

[16] Michael D. Bond, Milind Kulkarni, Man Cao, Meisam Fathi Salmi, and Jipeng Huang. 2015. Efficient Deterministic Replay of Multithreaded Executions in a Managed Language Virtual Machine. In *Proceedings of the Principles and Practices of Programming on The Java Platform (PPPJ '15)*. ACM, New York, NY, USA, 90–101. https://doi.org/10.1145/2807426.2807434

[17] Brad Spengler. 2003. PaX: The Guaranteed End of Arbitrary Code Execution. https://grsecurity.net/PaX-presentation.ppt.

[18] Derek Bruening and Qin Zhao. 2011. Practical memory checking with Dr. Memory. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '11)*. IEEE Computer Society, Washington, DC, USA, 213–223. http://dl.acm.org/citation.cfm?id=2190025.2190067

[19] Bugzilla. 2013. "libtiff (gif2tiff): possible heapbased buffer overflow in readgifimage()". http://bugzilla.maptools.org/show_bug.cgi?id=2451.

[20] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. 1998. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *In Proceedings of the 7th USENIX Security Symposium*. 63–78.

[21] cppreference. 2017. Atomic operations library. http://en.cppreference.com/w/cpp/atomic.

[22] Heming Cui, Jiri Simsa, Yi-Hong Lin, Hao Li, Ben Blum, Xinan Xu, Junfeng Yang, Garth A. Gibson, and Randal E. Bryant. 2013. Parrot: A Practical Runtime for Deterministic, Stable, and Reliable Threads. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 388–405. https://doi.org/10.1145/2517349.2522735

[23] Heming Cui, Jingyue Wu, John Gallagher, Huayang Guo, and Junfeng Yang. 2011. Efficient Deterministic Multithreading Through Schedule Relaxation. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*. ACM, New York, NY, USA, 337–351. https://doi.org/10.1145/2043556.2043588

[24] David Devecsery, Michael Chow, Xianzheng Dou, Jason Flinn, and Peter M. Chen. 2014. Eidetic Systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, Berkeley, CA, USA, 525–540. http://dl.acm.org/citation.cfm?id=2685048.2685090

[25] Joseph Devietti, Jacob Nelson, Tom Bergan, Luis Ceze, and Dan Grossman. 2011. RCDC: A Relaxed Consistency Deterministic Computer. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, New York, NY, USA, 67–78. https://doi.org/10.1145/1950365.1950376

[26] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. 2002. ReVirt: Enabling Intrusion Analysis Through Virtual-machine Logging and Replay. *SIGOPS Oper. Syst. Rev.* 36, SI (Dec. 2002), 211–224. https://doi.org/10.1145/844128.844148

[27] Laura Effinger-Dean, Brandon Lucia, Luis Ceze, Dan Grossman, and Hans-J. Boehm. 2012. IFRit: Interference-free Regions for Dynamic Data-race Detection. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '12)*. ACM, New York, NY, USA, 467–484. https://doi.org/10.1145/2384616.2384650

[28] Frank Ch. Eigler. 2003. *Mudflap: pointer use checking for C/C++*. Red Hat Inc.

[29] Weining Gu, Z. Kalbarczyk, Ravishankar, K. Iyer, and Zhenyu Yang. 2003. Characterization of linux kernel behavior under errors. In *2003*

International Conference on Dependable Systems and Networks, 2003. Proceedings. 459–468. https://doi.org/10.1109/DSN.2003.1209956

[30] Zhenyu Guo, Xi Wang, Jian Tang, Xuezheng Liu, Zhilei Xu, Ming Wu, M Frans Kaashoek, and Zheng Zhang. 2008. R2: An application-level kernel for record and replay. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*. USENIX Association, 193–208.

[31] Niranjan Hasabnis, Ashish Misra, and R. Sekar. 2012. Light-weight bounds checking. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization (CGO '12)*. ACM, New York, NY, USA, 135–144. https://doi.org/10.1145/2259016.2259034

[32] Reed Hastings and Bob Joyce. 1991. Purify: Fast detection of memory leaks and access errors. In *In Proc. of the Winter 1992 USENIX Conference*. 125–138.

[33] Nima Honarmand, Nathan Dautenhahn, Josep Torrellas, Samuel T. King, Gilles Pokam, and Cristiano Pereira. 2013. Cyrus: Unintrusive Application-level Record-replay for Replay Parallelism. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. ACM, New York, NY, USA, 193–206. https://doi.org/10.1145/2451116.2451138

[34] Nima Honarmand and Josep Torrellas. 2014. Replay Debugging: Leveraging Record and Replay for Program Debugging. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14)*. IEEE Press, Piscataway, NJ, USA, 445–456. http://dl.acm.org/citation.cfm?id=2665671.2665737

[35] Derek R. Hower and Mark D. Hill. 2008. Rerun: Exploiting Episodes for Lightweight Memory Race Recording. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA '08)*. IEEE Computer Society, Washington, DC, USA, 265–276. https://doi.org/10.1109/ISCA.2008.26

[36] Jeff Huang, Peng Liu, and Charles Zhang. 2010. LEAP: lightweight deterministic multi-processor replay of concurrent java programs. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 207–216.

[37] Jeff Huang, Charles Zhang, and Julian Dolby. 2013. CLAP: Recording Local Executions to Reproduce Concurrency Failures. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 141–152. https://doi.org/10.1145/2491956.2462167

[38] Shiyou Huang, Bowen Cai, and Jeff Huang. 2017. Towards Production-Run Heisenbugs Reproduction on Commercial Hardware. In *2017 USENIX Annual Technical Conference*. USENIX Association, 403–415.

[39] Intel Corporation. 2012. Intel Inspector XE 2013. http://software.intel.com/en-us/intel-inspector-xe.

[40] Baris Kasikci, Benjamin Schubert, Cristiano Pereira, Gilles Pokam, and George Candea. 2015. Failure Sketching: A Technique for Automated Root Cause Diagnosis of In-production Failures. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. ACM, New York, NY, USA, 344–360. https://doi.org/10.1145/2815400.2815412

[41] Joseph Kulandai. 2011. Java Hashtable. http://javapapers.com/core-java/java-hashtable/.

[42] Lubomir Kundrak. 2007. Buffer overflow in bzip2's bzip2recover. https://bugzilla.redhat.com/show_bug.cgi?id=226979.

[43] Dongyoon Lee, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. 2012. Chimera: Hybrid Program Analysis for Determinism. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 463–474. https://doi.org/10.1145/2254064.2254119

[44] Dongyoon Lee, Benjamin Wester, Kaushik Veeraraghavan, Satish Narayanasamy, Peter M. Chen, and Jason Flinn. 2010. Respec: Efficient Online Multiprocessor Replay via Speculation and External Determinism. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*. ACM, New York, NY, USA, 77–90. https://doi.org/10.1145/1736020.1736031

[45] Kyu Hyung Lee, Dohyeong Kim, and Xiangyu Zhang. 2014. Infrastructure-Free Logging and Replay of Concurrent Execution on Multiple Cores. In *Proceedings of the 28th European Conference on ECOOP 2014 — Object-Oriented Programming - Volume 8586*. Springer-Verlag New York, Inc., New York, NY, USA, 232–256. https://doi.org/10.1007/978-3-662-44202-9_10

[46] Peng Liu, Xiangyu Zhang, Omer Tripp, and Yunhui Zheng. 2015. Light: Replay via Tightly Bounded Recording. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2015)*. ACM, New York, NY, USA, 55–64. https://doi.org/10.1145/2737924.2738001

[47] Tongping Liu and Emery D. Berger. 2011. SHERIFF: precise detection and automatic mitigation of false sharing. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications (OOPSLA '11)*. ACM, New York, NY, USA, 3–18. https://doi.org/10.1145/2048066.2048070

[48] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. 2011. Dthreads: efficient deterministic multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*. ACM, New York, NY, USA, 327–336. https://doi.org/10.1145/2043556.2043587

[49] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. 2016. DoubleTake: Fast and Precise Error Detection via Evidence-Based Dynamic Analysis. In *Proceedings of 38th International Conference on Software Engineering (ICSE'16)*. ACM, New York, NY, USA.

[50] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. 2005. Bugbench: Benchmarks for evaluating bug detection tools. In *In Workshop on the Evaluation of Software Defect Detection Tools*.

[51] Nuno Machado, Brandon Lucia, and Luís Rodrigues. 2015. Concurrency debugging with differential schedule projections. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 586–595.

[52] Ali José Mashtizadeh, Tal Garfinkel, David Terei, David Mazieres, and Mendel Rosenblum. 2017. Towards Practical Default-On Multi-Core Record/Replay. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. ACM, New York, NY, USA, 693–708. https://doi.org/10.1145/3037697.3037751

[53] Microsoft. 2007. What is the Staging Environment? https://msdn.microsoft.com/en-us/library/ms942990(v=cs.70).aspx.

[54] Pablo Montesinos, Luis Ceze, and Josep Torrellas. 2008. DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Efficiently. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA '08)*. IEEE Computer Society, Washington, DC, USA, 289–300. https://doi.org/10.1109/ISCA.2008.36

[55] Mozilla Corporation. 2017. rr: lightweight recording & deterministic debugging. http://rr-project.org/.

[56] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. 2008. Finding and Reproducing Heisenbugs in Concurrent Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, Berkeley, CA, USA, 267–280. http://dl.acm.org/citation.cfm?id=1855741.1855760

[57] George C. Necula Necula, McPeak Scott, and Weimer Westley. 2002. CCured: Type-Safe Retrofitting of Legacy Code. In *Proceedings of the Principles of Programming Languages*. 128–139.

[58] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation (PLDI '07)*. ACM, New York, NY, USA, 89–100. https://doi.org/10.1145/1250734.1250746

[59] Oracle Corporation. 2011. Sun Memory Error Discovery Tool (Discover). http://docs.oracle.com/cd/E18659_01/html/821-1784/gentextid-302.html.

[60] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. 2017. Engineering Record And Replay For Deployability. In *2017 USENIX Annual Technical Conference*. USENIX Association.

[61] parasoft Company. 2013. *Runtime Analysis and Memory Error Detection for C and C++*.

[62] Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H. Lee, and Shan Lu. 2009. PRES: probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP '09)*. ACM, New York, NY, USA, 177–192. https://doi.org/10.1145/1629575.1629593

[63] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. 2010. PinPlay: A Framework for Deterministic Replay and Reproducible Analysis of Parallel Programs. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '10)*. ACM, New York, NY, USA, 2–11. https://doi.org/10.1145/1772954.1772958

[64] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. 2005. Rx: Treating Bugs As Allergies—a Safe Method to Survive Software Failures. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles (SOSP '05)*. ACM, New York, NY, USA, 235–248. https://doi.org/10.1145/1095810.1095833

[65] James Reinders. 2013. "Processor Tracing". https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing.

[66] Michiel Ronsse and Koen De Bosschere. 1999. RecPlay: A Fully Integrated Practical Record/Replay System. *ACM Trans. Comput. Syst.* 17, 2 (May 1999), 133–152. https://doi.org/10.1145/312203.312214

[67] Michiel Ronsse and Koen De Bosschere. 1999. RecPlay: a fully integrated practical record/replay system. *ACM Trans. Comput. Syst.* 17, 2 (May 1999), 133–152. https://doi.org/10.1145/312203.312214

[68] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: a fast address sanity checker. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference (USENIX ATC'12)*. USENIX Association, Berkeley, CA, USA, 28–28. http://dl.acm.org/citation.cfm?id=2342821.2342849

[69] SQL Developers. 2017. How SQLite Is Tested. https://www.sqlite.org/testing.html.

[70] Talos. 2016. "Memcached Server SASL Autentication Remote Code Execution Vulnerability". https://www.talosintelligence.com/reports/TALOS-2016-0221/.

[71] Joseph Tucek, Shan Lu, Chengdu Huang, Spiros Xanthos, and Yuanyuan Zhou. 2007. Triage: Diagnosing Production Run Failures at the User's Site. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles (SOSP '07)*. ACM, New York, NY, USA, 131–144. https://doi.org/10.1145/1294261.1294275

[72] Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. 2011. DoublePlay: parallelizing sequential logging and replay. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems (ASPLOS XVI)*. ACM, New York, NY, USA, 15–26. https://doi.org/10.1145/1950365.1950370

[73] Yan Wang, Harish Patil, Cristiano Pereira, Gregory Lueck, Rajiv Gupta, and Iulian Neamtiu. 2014. DrDebug: Deterministic Replay Based Cyclic Debugging with Dynamic Slicing. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '14)*. ACM, New York, NY, USA, Article 98, 11 pages. https://doi.org/10.1145/2544137.2544152

[74] Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, and Zhiqiang Ma. 2010. Ad Hoc Synchronization Considered Harmful. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*. USENIX Association, Berkeley, CA, USA, 163–176. http://dl.acm.org/citation.cfm?id=1924943.1924955

[75] Qiang Zeng, Dinghao Wu, and Peng Liu. 2011. Cruiser: concurrent heap buffer overflow monitoring using lock-free data structures. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation (PLDI '11)*. ACM, New York, NY, USA, 367–377. https://doi.org/10.1145/1993498.1993541