DOUBLETAKE: Evidence-Based Dynamic Analysis

Charlie Curtsinger Emery D. Berger
School of Computer Science
University of Massachusetts Amherst
Amherst, MA 01003
{charlie,emery}@cs.umass.edu

Tongping Liu
Department of Computer Science
The University of Texas at San Antonio
San Antonio, TX 78249-0667
tongpingliu@cs.utsa.edu

Abstract

Dynamic analysis can be helpful for debugging, but is often too expensive to use in deployed applications. We introduce evidence-based dynamic analysis, an approach that enables extremely lightweight analyses for an important class of errors: those that can be forced to leave evidence of their existence. Evidence-based dynamic analysis lets execution proceed at full speed until the end of an epoch. It then examines program state to find evidence that an error occurred at some time during that epoch. If so, execution is rolled back and re-execution proceeds with instrumentation activated to pinpoint the error. We present DOUBLETAKE, a prototype evidence-based dynamic analysis framework. We demonstrate its generality by building analyses to find buffer overflows, memory use-after-free errors, and memory leaks. DOUBLETAKE is precise and efficient: its buffer overflow analysis runs with just 2% overhead on average, making it the fastest such system to date.

1. Introduction

Dynamic analysis tools are widely used to find bugs in applications. They are popular among programmers because of their precision—for many analyses, they report no false positives—and can pinpoint the exact location of errors, down to the individual line of code.

Perhaps the most prominent and widely used dynamic analysis tool for C/C++ binaries is Valgrind [30]. Valgrind's most popular use case, via its default tool, memcheck, is to check memory errors, including buffer overflows, use-after-free errors, and memory leaks.

Unfortunately, while these dynamic analysis tools are useful, they are often expensive. Using Valgrind typically slows down applications by 10-100×. Faster dynamic analysis frameworks exist for finding particular errors, but all impose substantial overheads. Google's AddressSanitizer, for example, detects buffer overflows and use-after-free errors, but slows applications by around 30% [39]. Precise memory leak detectors that identify the point at which objects are leaked remain far more expensive.

Because of their overhead, this class of dynamic analysis tools are only used during debugging. However, they are limited by definition to the executions that they have seen. The fact that using these tools in deployed applications is not practical means that errors that could have been found trivially instead require painstaking debugging later.

This paper presents a new approach that enables extremely lightweight dynamic analysis for an important class of errors. These errors share a monotonicity property: when an error happens, evidence that it happened either remains or grows in memory so that it can be recovered at a later point. When this evidence is not naturally occurring, it is often possible to "plant" evidence via what we call *tripwires* to ensure later detection. An example tripwire is a random value, also known as a "canary", placed in unallocated space between heap objects [13]. A corrupted canary is incontrovertible evidence that a buffer overflow occurred at some time in the past.

We present an approach called *evidence-based dynamic analysis* that is based on the following key insight: by combining checkpointing with evidence gathering, it is possible to let applications run at full speed in the common case (no errors). If we discover evidence of an error, we can go back and re-execute the program with instrumentation activated to find the exact cause of the error.

We present a prototype evidence-based dynamic analysis framework called DOUBLETAKE. DOUBLETAKE performs its checkpoints only at irrevocable system calls, amortizing the cost of checkpoint collection. Each checkpoint saves the contents of the stack, globals, registers, and the heap. If it finds evidence of an error at the next system call or after a segmentation violation, DOUBLETAKE re-executes the application from the most recent checkpoint. During re-execution, it triggers instrumentation to let it precisely locate the source of the error. For buffer overflows, DOUBLETAKE sets hardware watchpoints on the tripwire memory locations that were found to be corrupted. During re-execution, DOUBLETAKE can pinpoint exactly the point where the buffer overflow occurred.

We implement DOUBLETAKE as a drop-in library that can either be linked directly with the application under analysis, or which can be activated by setting an environment variable (`LD_PRELOAD` on Unix systems) to dynamically load DOUBLETAKE before execution. No re-compilation or availability of source code is required. This approach makes DOUBLETAKE as convenient to use as Valgrind.

We have built three different analyses using DOUBLETAKE: buffer overflow detection, use-after-free detection, and memory leak detection. All of these analyses run without any false positives, precisely pinpoint the error location, and operate with *extremely* low overhead: for example, with DOUBLETAKE, buffer overflow analysis operates with just 2% overhead on average, making it the fastest overflow detector to date and thus feasible to use in deployed scenarios.

Contributions

The contributions of this paper are the following:

1. It introduces *evidence-based* dynamic analysis, a new analysis technique that combines checkpointing with evidence gathering and instrumented replay to enable precise error detection with extremely low overhead.
2. It presents DOUBLETAKE, a framework that implements evidence-based dynamic analyses for C/C++ programs: its analyses (detecting buffer overflows, use-after-frees, and memory leaks) are the fastest reported to date.

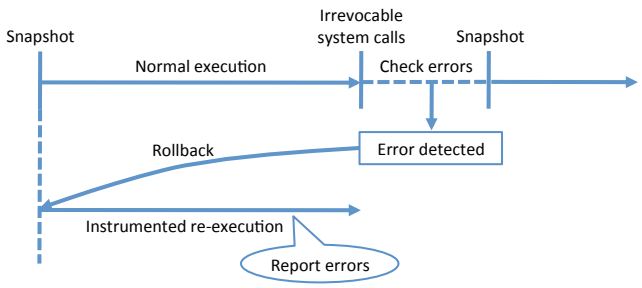

Figure 1. Overview of DOUBLETAKE: execution is divided into epochs at the boundary of irrevocable system calls.

2. Overview

DOUBLETAKE is a high performance dynamic analysis framework for a class of errors that share a *monotonicity* property: evidence of the error is persistent and can be gathered after-the-fact. As Figure 1 depicts, program execution is divided into epochs, during which execution proceeds at full speed. At the end of each epoch, marked by irrevocable system calls, DOUBLETAKE checks program state for evidence of memory errors. If an error is found, the epoch is re-executed with additional instrumentation to pinpoint the exact cause of the error. To demonstrate DOUBLETAKE's effectiveness, we have implemented detection tools for heap buffer overflows, use-after-free errors, and memory leaks, which we describe in detail in Section 3. All detection tools share the following core infrastructure that DOUBLETAKE provides.

2.1 Efficient Recording

At the beginning of every epoch, DOUBLETAKE saves a snapshot of program registers, and all writable memory. The epoch ends when the program attempts to issue an irrevocable system call, but most system calls do not end the current epoch. DOUBLETAKE also records the order of thread synchronization operations to support re-execution of parallel programs. DOUBLETAKE records minimal system state at the beginning of each epoch (like file offsets), which allows system calls that modify this state to be undone if re-execution is required. As a result, most programs require very few epochs and program state checks. We describe the details of each application's state checks in Section 3.

2.2 Precise Replay

When program state checks detect an error, DOUBLETAKE replays the previous epoch to pinpoint the error's root cause. DOUBLETAKE ensures that all program-visible state, system call results, memory allocations, and the order of all thread synchronization operations are identical to the original run. During replay, DOUBLETAKE returns saved return values for most system calls, with special handling for some cases. Section 4 describes DOUBLETAKE's recording and re-execution of system calls and synchronizations.

2.3 Custom Heap Allocator

DOUBLETAKE replaces the default heap allocator with a new heap built using Heap Layers [4]. Detection tools can interpose on heap operations to alter memory allocation requests or defer reuse of freed memory, and can mark the status of each object on its header. Memory leak detection uses this information to identify unreachable memory. Buffer overflow and use-after-free detection both use heap canaries to detect errors. DOUBLETAKE includes a bitmap to track the locations of heap canaries, and automatically checks the state of canaries at the end of each epoch. Section 4.2 presents further details.

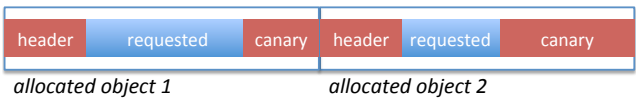

Figure 2. Heap organization used to provide evidence of buffer overflow errors. Object headers and unrequested space within allocated objects are filled with canaries; a corrupted canary indicates an overflow occurred.

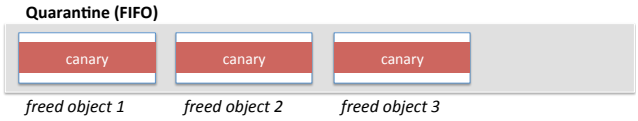

Figure 3. Evidence-based detection of dangling pointer (use-after-free) errors. Freed objects are deferred in a quarantine in FIFO order and filled with canaries. A corrupted canary indicates that a write was performed after an object was freed.

2.4 Watchpoints

DOUBLETAKE lets detection tools set hardware watchpoints during re-execution. A small number of watchpoints are available on modern architectures (four on x86). Each watchpoint can be configured to pause program execution when a specific byte or word of memory is accessed. Watchpoints are primarily used by debuggers, but previous approaches have used watchpoints for error detection as well [10, 11]. DOUBLETAKE's watchpoints are particularly useful in combination with heap canaries. During re-execution, our buffer overflow and use-after-free detectors place a watchpoint at the location of the overwritten canary to trap the instruction(s) responsible for the error.

3. Applications

We have implemented three error detection tools with DOUBLETAKE. These applications are intended to demonstrate DOUBLETAKE's generality and efficiency. The mechanisms used here are similar to those used by existing tools, but DOUBLETAKE runs with substantially lower overhead than prior tools.

3.1 Heap Buffer Overflow Detection

Heap buffer overflows occur when programs write outside the bounds of an allocated object. Our buffer overflow detector places canaries between heap objects, and reports an error when canary values have been overwritten. If an overwritten canary is found, the detector uses watchpoints during re-execution to identify the instruction responsible for the overflow.

Detection

Figure 2 presents an overview of the approach used to locate buffer overflows. All heap objects of DOUBLETAKE are managed by power of two size, adding two words for canaries (16 bytes) for each heap object. For an allocation size not power of two size (a non-aligned object), DOUBLETAKE rounds it up to the next power of two size class, putting byte-based canaries and word-based canaries immediately after this object. This approach lets DOUBLETAKE catch overflows as small as one byte. At memory deallocation, DOUBLETAKE checks buffer overflow for non-aligned objects, with size different with the power of two class. At the end of each epoch, DOUBLETAKE checks whether any canaries before and after aligned objects have been overwritten. For those non-aligned objects, DOUBLETAKE also checks buffer overflows be-

fore an object is freed. If an overwritten canary is found, a buffer overflow has been detected and re-execution begins.

Re-Execution

DOUBLETAKE installs a watchpoint at the address of the corrupted canary before re-execution. When the program is re-executed, any instruction that writes to this address will trigger the watchpoint. The operating system will deliver a `SIGTRAP` signal to DOUBLETAKE before the instruction is executed. By handling this signal, DOUBLETAKE reports the call stack of trapped instructions, obtained by using `backtrace` function.

3.2 Use-After-Free Detection

Use-after-free or dangling pointer overflow errors occur when an application continues to access memory through pointers that have been passed to `free()` or `delete`. Writes to freed memory can overwrite the contents of other live objects, leading to unexpected program behavior. Like the buffer overflow detector, our use-after-free detector uses canaries to detect writes to freed memory. When a use-after-free error is detected, DOUBLETAKE reports the allocation and deallocation sites of the object, and all instruction(s) that wrote to the object after it was freed.

Detection

Fiure 3 illustrates how to detect use-after-free. The use-after-free detector delays the re-allocation of freed memory. Freed objects are placed in a FIFO quarantine list, the same mechanism used by AddressSanitizer [39]. Objects are released from the quarantine list when the total size of quarantined objects exceeds 16 megabytes, or when there are 1,024 quarantined objects. This threshold is easily configurable. Objects in the quarantine list are filled with canary values, up to 128 bytes. Filling large objects entirely with canaries introduces too much overhead during normal execution, and is unlikely to catch any additional errors.

Before an object can be returned to the program heap, DOUBLETAKE verifies that no canaries has been overwritten. All canaries are checked at epoch boundaries. If a canary has been overwritten, the detector raises an error and begins re-execution.

Re-Execution

During re-execution, the use-after-free detector interposes on `malloc` and `free` calls to find the allocation and deallocation sites of the overwritten object. The detector records a call stack for both sites using the `backtrace` function. After the object is freed, the detector installs a watchpoint at the address of the overwritten canary. As with buffer overflow detection, any writes to the watched address will generate a `SIGTRAP` signal. The last allocation and deallocation sites, and the instructions responsible for the use-after-free error are reported at the end of re-execution.

3.3 Memory Leak Detection

Heap memory is leaked when it becomes inaccessible without being freed. Memory leaks can significantly degrade program performance due to increased memory footprint. Our leak detector identifies possible unreachable allocated objects at the end of each epoch. Allocation sites can help users fix memory leaks, but collecting these information for all`malloc` calls would unnecessarily slow down the program in normal execution for the common case (no memory leaks). Instead, DOUBLETAKE only records the allocation sites of leaked memory during re-execution, and adds no overhead for normal execution.

Detection

We detect memory leaks using the same marking mechanism as conservative garbage collection [44]. The marking phase performs a breadth-first scan of reachable memory using a work queue. Initially, all values in registers, globals, and the stack that look like pointers are added to the work queue. Any eight-byte aligned value that falls within the range of allocated heap memory is treated as a pointer.

At each step in the scan, the detector takes the first item off the work queue. Using the heap metadata located before each object, the detector finds the bounds of a object. Every object is allocated with a header containing "marked" and "allocated" bits. If the "marked" bit is set, the detector removes this object and moves on to the next item in the queue. If the object is allocated and unmarked, the detector sets the "marked" bit and adds all pointer values within the object's bounds to the work queue. When the work queue is empty, DOUBLETAKE ends its scan.

DOUBLETAKE then traverse the whole heap to find those allocated but unmarked objects. They are identified as memory leaks. If there are memory leaks, re-execution begins. The detector can also find potential dangling pointers (reachable freed objects). This option is disabled by default because, unlike other applications, potential dangling pointer detection could produce false positives.

Re-Execution

During re-execution, the leak detector checks the results of each `malloc` call. When the allocation of a leaked object is found, the detector records the call stack using the `backtrace` function. At the end of the epoch re-execution, the detector reports the last call stack for each leaked object.

4. Implementation

DOUBLETAKE is implemented as a library and can be linked directly to programs, or can be injected into unmodified binaries by setting the `LD_PRELOAD` environment variable on Linux.

At startup, DOUBLETAKE begins a new epoch. The epoch continues until the program issues an *irrevocable* system call (see Section 4.2 for details). Before this call is issued, DOUBLETAKE checks program state for evidence of errors. The details are presented in Section 3.

If no errors are found, DOUBLETAKE ends the epoch, issues the irrevocable system call, and begins a new epoch. If it has found evidence of an error, DOUBLETAKE enters re-execution mode. The remainder of this section describes the implementation of DOUBLETAKE's core functionality.

4.1 Epoch Start

At the beginning of each epoch, DOUBLETAKE takes a snapshot of program state. DOUBLETAKE saves all writable memory (stack, heap, and globals) from the main program and any linked libraries, and saves register state of each thread with the `getcontext` function. Read-only memory does not need to be saved. To identify all writable mapped memory, DOUBLETAKE reads the Linux `/proc/self/map` file. DOUBLETAKE also saves file positions of all open files. This lets programs issue `read` and `write` system calls without ending the current epoch. DOUBLETAKE uses the saved memory state and file positions to "undo" these calls if the epoch needs to be re-executed when an error is found.

4.2 Normal Execution

Once a snapshot has been saved, DOUBLETAKE lets the program execute normally. Most program operations proceed normally, but DOUBLETAKE interposes on heap allocations and system calls in order to set tripwires and support re-execution.

Category	Functions
Repeatable	`getpid`, `sleep`, `pause`
Recordable	`mmap`, `gettimeofday`, `time`, `clone` , `open`
Revocable	`write`, `read`
Deferrable	`close`, `munmap`
Irrevocable	`fork`, `exec`, `exit`, `lseek`, `pipe`, `flock`, `socket related system calls`

Table 1. System calls handled by DOUBLETAKE. All unlisted system calls are conservatively treated as irrevocable, and will end the current epoch. Section 4.2 describes how DOUBLETAKE handles calls in each category.

System Calls

DOUBLETAKE ends each epoch when the program attempts to issue an irrevocable system call. However, most system calls can safely be re-executed or undone prior to re-execution.

DOUBLETAKE breaks system calls into five categories, shown in Table 1. System calls could be intercepted using `ptrace`, but this would add unacceptable overhead during normal execution. Instead, DOUBLETAKE interposes on all library functions that may issue system calls.

Repeatable system calls do not modify system state, and return the same result during normal execution and re-execution. No special handling is required for these calls. CCIf you don't guarantee perfect replay, then you must argue that perfect replay isn't necessary and that at the very least the attempted replay does not affect application behavior.

Recordable system calls may return different results if they are re-executed. DOUBLETAKE records the result of these system calls during normal execution, and returns the saved result during re-execution. Some recordable system calls, such as `mmap`, change the state of underlying operating system.

Revocable system calls modify system state, but DOUBLETAKE can save the original state beforehand and restore it prior to re-execution. Most file I/O fall into this category. For example, `write` modifies file contents, DOUBLETAKE can write the same content during re-execution. `write` also changes the current file position, which DOUBLETAKE restores to the saved file position using `lseek` prior to re-execution. DOUBLETAKE saves all file descriptors of opened files in a hash table at the beginning of each epoch. In addition, DOUBLETAKE must save stream contents returned by `fread`. Calls to `read` and `write` on normal files, which can be identified by check the hash map, don't need to be handled. But those calls on socket files are treated as irrevocable system calls.

Deferrable system calls will irrevocably change program state, but can safely be delayed until the end of the current epoch. DOUBLETAKE delays all calls to `munmap` and `close`, and executes these system calls before exiting or starting a new epoch.

Irrevocable system calls change internally-visible program state, and cannot be undone. DOUBLETAKE must end the current epoch before these system calls are allowed to proceed. Note that for DOUBLETAKE, the meaning of "irrevocable" is different from that used in transactional memory systems [43]. Unlike in transactions, we expect re-execution to be identical to the epoch's original execution. It is safe for system calls to affect externally-visible state as long as the effect on internal state can be hidden or undone.

Multithreaded Support

We have implemented support for multiple threads, but the recording and re-execution of thread synchronizations is not yet stable.

DOUBLETAKE records the sequence of synchronizations and corresponding results separately for each thread. Basically, a synchronization event is recorded in two lists simultaneously, a per-thread list and a synchronization variable list.

Every mutex records the order of threads that acquire it, and condition variables record the order of thread wake-ups. DOUBLETAKE does not enforce a total global order on lock acquisitions. Operations within a single thread are totally-ordered and recorded, and DOUBLETAKE enforces local order . In the absence of data races, this is sufficient to ensure deterministic re-execution. If it fails to reveal the error on replay, DOUBLETAKE has effectively tolerated the error and continues execution.

Calls to `pthread_create` are recorded with the same mechanism as recordable system calls. When a new thread starts, DOUBLETAKE takes a snapshot of the thread's stack and registers to enable re-execution from the beginning of the thread's execution. As with synchronization operations, DOUBLETAKE logs thread creation order and enforces this order during re-execution. Threads exits are deferred until the end of the epoch. `pthread_join` is effectively deferred as well.

Heap Allocator

Heap allocators typically issue a large number of `mmap` or `sbrk` system calls, which would complicate DOUBLETAKE's logging to facilitate re-execution. DOUBLETAKE replaces the default heap with a fixed-size BiBOP-style allocator with per-thread subheaps and power-of-two size classes, built using Heap Layers [4].

DOUBLETAKE uses a separate heap for internal usage and only uses this custom heap allocator for allocations from the application and corresponding libraries. Any additional memory allocations in the replay phase are also allocated from a separate heap too, such as the `backtrace` call or printing.

DOUBLETAKE's heap is completely deterministic for a program without races, so no logging is required to ensure that allocations do not change during re-execution. To achieve this target, all allocations from the same thread is met at a specific subheap, which obtains every superblock (large chunks of memory) by acquiring a lock. Superblock allocations are deterministic in replay phase, which is enforced by the deterministically acquiring and releasing of a lock. When an object is freed, this object is returned to the current thread issuing this free operation. Because of deterministic superblock allocations and memory free operations (enforced by the program order), we guarantee that all memory allocations are deterministic inside the same subheap.

DOUBLETAKE keeps information about each heap object in its object header. During allocation, objects are marked as allocated and their *requested size* are saved, which may be less than the power-of-two size class. At deallocation, objects are marked as freed and placed into the quarantine list if the detection of use-after-free is enabled. For non-aligned objects, DOUBLETAKE also checks buffer overflows for non-aligned words, which can even report one-byte buffer overflows because of exact size.

DOUBLETAKE also maintains a bitmap to record the locations of heap canaries. The bitmap records every word of heap memory that contains a canary. DOUBLETAKE notifies the detection tool when any of the bytes do not contain canaries. Buffer overflow detection places canaries only outside the requested object size. Re-execution is only started if the detection tool finds that canaries between allocated objects have been overwritten.

During replay, DOUBLETAKE's heap allocator checks to see if the object being allocated or freed contains the address where an error was detected. If so, DOUBLETAKE calls the `backtrace()` function to obtain a call stack for the allocation and deallocation sites.

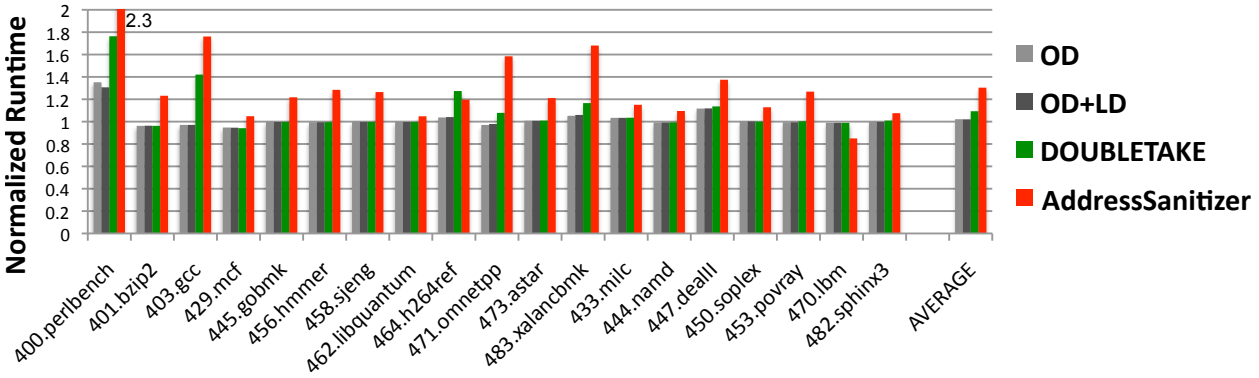

Figure 4. Runtime overhead of DOUBLETAKE (OD = Buffer Overflow Detection, LD = Leak Detection, DOUBLETAKE = all three detections enabled) and AddressSanitizer, normalized to each benchmark's original execution time.

Epoch End

The epoch ends when any thread issues an irrevocable system call. All other threads are notified with a signal SIGUSR2. Once all threads have stopped, DOUBLETAKE checks the program state for errors. The application-specific error checks are described in Section 3. If an error is found, DOUBLETAKE immediately switches to re-execution mode. If not, the runtime issues any deferred system calls and clears the logs for all recorded system calls.

4.3 Rollback

Before re-executing the current epoch, DOUBLETAKE must roll back the program state.

Restoring the saved stack may corrupt the current stack, so DOUBLETAKE switches to a temporary stack during rollback. The saved state of all writable memory is copied back, and revocable system calls are undone by calling `lseek` to reset all opening files positions. Then DOUBLETAKE installs watchpoints before calling `setcontext` to restore its context.

Debug registers are not accessible in user-mode, so DOUBLETAKE must use `ptrace` to set watchpoints. DOUBLETAKE forks a child process and attaches to it using `ptrace` to load watched addresses into the debug registers and enable the watchpoints.

Once watchpoints have been placed, DOUBLETAKE uses the `setcontext` call to restore register state and begin re-execution.

4.4 Re-Execution

During re-execution, DOUBLETAKE replays the saved results of recordable system calls from the log collected during normal execution. All deferred system calls are converted to no-ops while the program is re-executing. DOUBLETAKE issues other system calls normally.

Synchronization Replay

DOUBLETAKE enforces the recorded order of synchronization operations during re-execution by using the *semaphore replay* mechanism [14]. DOUBLETAKE assigns a binary semaphore for each thread, initialized as unavailable. Lock acquisitions and conditional wakeups are treated similarly here.

Before the rollback, DOUBLETAKE checks the first synchronization event of every synchronization variable list. If an synchornization event is also the first event of a thread, the semaphore of this thread is incremented immediately. Otherwise, this thread are granted with a pending increment, incremented only after all previous synchronizations of this thread has been handled.

Benchmark	Overhead	Benchmark	Overhead
400.perlbench	20.5×	458.sjeng	20.3×
401.bzip2	16.8×	471.omnetpp	13.9×
403.gcc	18.7×	473.astar	11.9×
429.mcf	4.5×	433.milc	11.0×
445.gobmk	28.9×	444.namd	24.9×
456.hmmer	13.8×	450.dealII	42.8×

Table 2. Valgrind runtime overhead.

In the replay, a lock acquiring is turned into a semaphore wait. During a lock release, DOUBLETAKE actually increments the semaphore of next thread in the same synchronization variable list, no matter whether the next thread is the same thread or not. In order to support multiple locks for a critical section, a lock acquisition also increments pending increments if possible.

4.4.1 Trap Handler

DOUBLETAKE handles traps caused by accesses on watchpoints. Inside the trap handler, DOUBLETAKE firstly determines which watchpoint causes the current trap if the fault instruction is not issued from DOUBLETAKE library. Normally, the debug status register has to be accessed in order to know this information. However, using `ptrace` is very inconvenient inside a signal handler because of involving in another process. DOUBLETAKE always keeps an updated value for every watchpoint. Thus, it can precisely determine which watchpoint is triggered by checking the changes of those watchpoints. Also, DOUBLETAKE prints callsite information of buffer overflows or use-after-free errors and their memory allocation (or deallocation).

5. Evaluation

We evaluate DOUBLETAKE to demonstrate its efficiency, both in runtime and memory overhead. We also demonstrate the effectiveness of our detection tools across a benchmark suite and with several real applications. All experiments are run on a quiescent Intel Core 2 dual-processor system with 16GB of RAM running Linux 2.6.18, and version 2.5 of `glibc`. Each processor is a 4-core 64-bit Intel Xeon, operating at 2.33GHz with a 4MB shared L2 cache a 32KB per-core L1 cache. All benchmarks are built as 64-bit executables using LLVM 3.2 with the clang front-end and `-O2` optimizations.

Benchmark	Processes	Epochs	Syscalls	Mallocs (#)
400.perlbench	3	291	60068	360605640
401.bzip2	6	6	968	168
403.gcc	9	9	155505	28458514
429.mcf	1	1	24443	5
445.gobmk	5	5	2248	658034
456.hmmer	2	2	46	2474268
458.sjeng	1	1	23	5
462.libquantum	1	1	11	179
464.h264ref	3	885	2592	146827
471.omnetpp	1	1	19	267168472
473.astar	2	2	102	4799955
483.xalancbmk	1	1	123706	135155557
433.milc	1	1	12	6517
444.namd	1	1	470	1324
447.dealII	1	1	8131	151332314
450.soplex	2	2	37900	310619
453.povray	1	1	25721	2461141

Table 3. Benchmark characteristics.

5.1 Runtime Overhead

The runtime and memory overhead of DOUBLETAKE is evaluated with all C and C++ SPEC CPU2006 benchmarks, 19 in total. We compare DOUBLETAKE with AddressSanitizer and Valgrind. AddressSanitizer is the previous state-of-the-art for detecting buffer overflows and use-after-free errors, but cannot detect memory leaks [39]. Valgrind's Memcheck tool is widely used to detect buffer overflows, memory leaks, and use-after-free errors [30].

During performance evaluation, we disable DOUBLETAKE's rollback to measure only the overhead of normal execution. DOUBLETAKE's memory error detectors only run on the heap, so AddressSanitizer is configured to check only writes to heap memory. For each benchmark, we report the average of three runs with the largest input size, except for Valgrind. We only run Valgrind once because of its high runtime overhead.

Figure 4 shows the runtime overhead results for DOUBLETAKE and AddressSanitizer. Results for Valgrind do not fit on the graph, and are presented separately in Table 2. On average, DOUBLETAKE adds only 9% overhead *with all three error detectors enabled*. Without use-after-free detection, DOUBLETAKE's overhead is just 3%. Overflow detection alone slows execution by just 2%. AddressSanitizer has an average runtime overhead of 30%. Valgrind has an average runtime overhead of 20×, but two benchmarks (`perlbench` and `sjeng`) have not yet finished running.

For 17 out of 19 benchmarks, DOUBLETAKE outperforms AddressSanitizer, even with memory leak detection enabled. For 12 benchmarks, DOUBLETAKE's runtime overhead with all detectors enabled is under 3%. Both DOUBLETAKE and AddressSanitizer substantially outperform Valgrind on all benchmarks. Three benchmarks, `400.perlbench`, `403.gcc` and `447.dealII`, have substantially higher overhead than most benchmarks with both DOUBLETAKE and AddressSanitizer. Table 4 shows that DOUBLETAKE and AddressSanitizer both add substantial memory overhead for these benchmarks. This increased memory footprint is likely responsible for the degraded performance due to increased cache and TLB pressure.

DOUBLETAKE's use-after-free detection adds roughly 6% runtime overhead: only `perlbench`, `gcc`, and `h264ref` run with more than 20% overhead. As described in Section 3.2, all freed objects are filled with canaries (up to 128 bytes). DOUBLETAKE spends a substantial amount of time filling freed memory with canaries for applications with a large number of `malloc` and `free` calls.

Table 3 shows detailed benchmark characteristics. The "Processes" column shows the number of different invocations in the input set. The number of epochs is significantly lower than the number of system calls because of DOUBLETAKE's lightweight system call handling. Benchmarks with the highest overhead run a substantial number of epochs (`perlbench` and `h264ref`) and make a large number of `malloc` calls (`gcc`, `omnetpp`, and `xalancbmk`).

5.2 Memory Overhead

Much of DOUBLETAKE's memory overhead comes from the snapshot of writable memory at the beginning at each epoch. However, the first snapshot is very small because the heap is completely empty. The benchmarks `bzip2`, `mcf`, `sjeng`, `milc`, and `lbm` run in a single epoch, and therefore have very low memory overhead. System call logs introduce a small amount of additional overhead. Other sources of memory overhead are application-specific: buffer overflow detection adds space between heap objects, which can increase memory usage for programs with many small allocations. Use-after-free detection adds constant-size memory overhead by delaying memory reuse.

Figure 5 shows memory overhead for DOUBLETAKE and AddressSanitizer, and Table 4 contains a detailed breakdown. We measure program memory usage by recording the peak *physical* memory usage because DOUBLETAKE's pre-allocated heap consumes 4GB of virtual memory. Peak memory usage is collected by periodically sampling the proportional set size files (`/proc/self/smaps`).

On average, DOUBLETAKE imposes 2.8× memory overhead, while AddressSanitizer introduces 4.8× overhead. For `povray` and `h264ref`, both tools introduce large relative memory overhead because these benchmarks use just 3MB and 24MB respectively. Complete memory usage results are shown in Table 4. For all other benchmarks, both DOUBLETAKE and AddressSanitizer introduce less than 5× memory overhead. DOUBLETAKE has lower memory overhead than AddressSanitizer on all but two benchmarks: `perlbench` and `namd`. DOUBLETAKE's total memory usage is less than twice that of the original programs, and 20% less than AddressSanitizer.

Benchmark	Original	AddressSanitizer	DOUBLETAKE
400.perlbench	656	1481	1977
401.bzip2	870	1020	1003
403.gcc	683	2293	1583
429.mcf	1716	1951	1994
445.gobmk	28	137	58
456.hmmer	24	256	129
458.sjeng	179	220	203
462.libquantum	66	144	131
464.h264ref	65	179	247
471.omnetpp	172	538	291
473.astar	333	923	477
483.xalancbmk	428	1149	801
433.milc	695	1008	917
444.namd	46	79	92
447.dealII	514	2496	1727
450.soplex	441	1991	1654
453.povray	3	133	50
470.lbm	418	496	470
482.sphinx3	45	181	98
Total	**7386**	**16678**	**13906**

Table 4. DOUBLETAKE and AddressSanitizer memory use (MB).

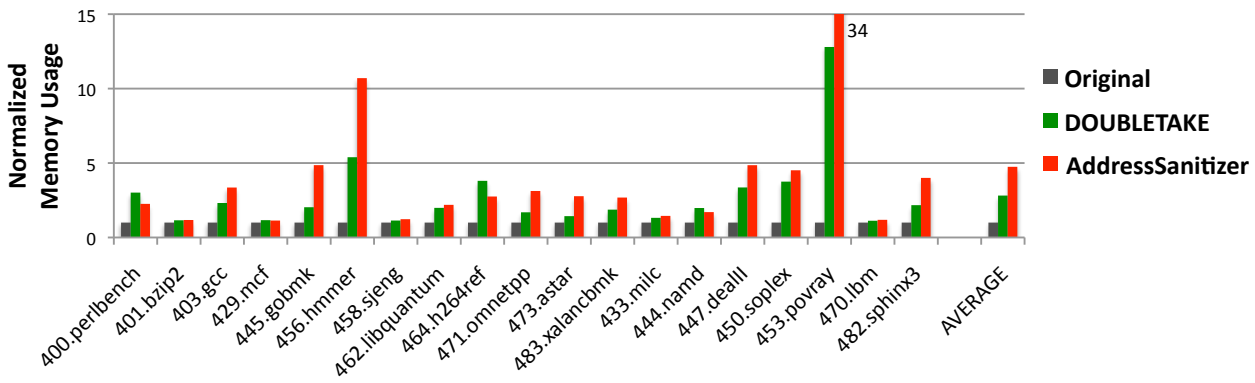

Figure 5. Memory overhead of DOUBLETAKE and AddressSanitizer.

5.3 Effectiveness

We use DOUBLETAKE to find errors in both the SPEC CPU2006 benchmark suite and a suite of real applications.

Benchmarks. DOUBLETAKE detected a one-byte heap buffer overflow in `perlbench`, which is not detected by AddressSanitizer. DOUBLETAKE also detected a significant number of memory leaks in `perlbench` and `gcc`, which we have verified using Valgrind's Memcheck tool.

Real applications. DOUBLETAKE detects known buffer overflows in `libHX`, `bzip2`, `vim`,`bc`, `polymorph`, and `gzip`. Buggy inputs were obtained from prior buffer overflow detection tools, Red Hat's Bugzilla, and bugbench [19, 22, 26, 45].

The buffer overflows we observed in these applications are triggered by specific inputs, which are difficult to detect during development. In most cases, DOUBLETAKE's overhead is low enough to be enabled in deployment, which would make it possible to detect these bugs in the field. DOUBLETAKE also detects memory leaks in `gcc-4.4.7` and `vim`.

6. Discussion

In this section, we discuss DOUBLETAKE's limitations, the limitations of three detection tools we have implemented, and our plans for future work on DOUBLETAKE.

6.1 Limitations

DOUBLETAKE lets dynamic analyses run with very low overhead, then re-execute with heavyweight instrumentation when an error has been detected. This approach will not work for errors that are not monotonic: once an invariant has been violated, its evidence must remain until the end of the epoch.

Applications

We have used DOUBLETAKE to implement detectors for heap buffer overflows, use-after-free errors, and memory leaks. These tools demonstrate DOUBLETAKE's effectiveness as a framework for efficient dynamic analyses, but they are not perfect. Our heap buffer overflow detector cannot identify all non-contiguous buffer overflows. If an overflow touches memory only in adjacent objects and not canaries, DOUBLETAKE's end-of-epoch scan will not find any evidence of the overflow. Both the buffer overflow and use-after-free detectors can detect errors only on writes. To reduce overhead, use-after-free detection only places canaries in the first 128 bytes of freed objects. If a write to freed memory touches only above this threshold, our detector will not find it. The memory leak detector will not produce false positives, but non-pointer values that look like pointers to leaked objects can lead to false negatives. Finally, if a leaked object was not allocated in the current epoch, DOUBLETAKE's re-execution will not be able to find the object's allocation site. In practice, DOUBLETAKE's epochs are long enough to collect allocation site information for all leaks detected during our evaluation.

6.2 Future Work

We plan to explore further analyses directions, including concurrency error detection.

To further reduce DOUBLETAKE's overhead, we plan to replace DOUBLETAKE's custom allocator with a more efficient heap. For use-after-free detection, large objects could be allocated directly using `mmap` and protected rather than filling them with canaries.

We also plan to integrate the DOUBLETAKE framework with `gdb`. Lightweight rollback and re-execution would be useful for diagnosing application errors during a debugging session. Additionally, a program run with DOUBLETAKE could automatically begin a `gdb` session when an error is detected.

7. Related Work

In this section, we discuss related approaches to dynamic analysis and efficient record and replay systems.

7.1 Dynamic Analysis

Dynamic analyses typically rely on one or more of the following approaches: dynamic instrumentation, static instrumentation, and interposition. We discuss prior analysis tools below, grouped by approach.

Dynamic Instrumentation

A large number of error detection tools use dynamic instrumentation, including many commercial tools. Valgrind's Memcheck tool, Dr. Memory, Purify, Intel Inspector, and Sun Discover all fall into this category [8, 17, 20, 30, 34]. These tools use dynamic instrumentation engines, such as Pin, Valgrind, and DynamiRIO [7, 27, 30]. These tools can detect memory leaks, use-after-free errors, uninitialized reads, and buffer overflows. Dynamic instrumentation tools are typically easy to use because they do not require recompilation, but this easy of use comes at the cost of high overhead. Programs run with Valgrind take 20× longer than usual, and Dr. Memory introduces 10× runtime overhead. DOUBLETAKE is *significantly* more efficient than prior dynamic instrumentation tools, with under 10% performance overhead.

Static Instrumentation

Static instrumentation-based techniques leverage compiler analyses and efficient code generation to implement more efficient dynamic analyses tools. Mudflap instruments references through pointers to detect buffer overflows, invalid heap usage, and memory leaks [15]. AddressSanitizer, CCured, LBC, Insured++, and Baggy bounds-checking leverage static analysis to reduce the amount of instrumentation [1, 16, 29, 39]. Compared to dynamic instrumentation-based approaches, these tools incur substantially lower overhead than dynamic instrumentation-based approaches, but are more difficult to use. Tools that rely on static instrumentation cannot detect errors in code that was not recompiled with instrumentation. DOUBLETAKE is more efficient than AddressSanitizer, the previous state-of-the-art, and enables dynamic analysis on the entire program (including libraries) with no recompilation.

Interposition

DOUBLETAKE uses library interposition exclusively during normal execution. More expensive instrumentation is only introduced after an error has been detected. BoundsChecker interposes on Windows heap library calls to detect memory leaks, use-after-free errors and buffer overflows [28]. Many prior approaches use a mix of library interposition and virtual memory techniques to detect memory errors [3, 5, 9, 18, 23, 25, 32, 33, 36, 45]. Unlike DOUBLETAKE, the overhead of these systems remains high.

7.2 Record-and-Replay

Record and replay systems have broad applications. There are numerous replay-based approaches for software debugging and fault tolerance [6, 21, 37, 38, 40, 41]. Flashback records the results of every system call to facilitate deterministic replay, but with higher overhead than DOUBLETAKE [40]. DOUBLETAKE records only the necessary system state, and "undoes" the effect of system calls before re-execution. Triage automates the failure diagnosis process, but must be activated manually when an error is observed [41]. Aftersight and Speck use record and replay for dynamic analysis, but incur substantially higher overhead than DOUBLETAKE [12, 31]. Aftersight executes programs in a virtual machine, and records all inputs to the VM. Speck is focused on security checks, including data-flow tracking and virus scanning: applications that likely require the always-on instrumentation that DOUBLETAKE does not provide. Other systems have focused on reducing the performance overhead of recording [2, 24, 35, 42]. None of these systems is as efficient as DOUBLETAKE.

8. Conclusion

This paper introduces *evidence-based dynamic analysis*, a new lightweight dynamic analysis technique. Evidence-based dynamic analysis works for errors that can be forced to leave evidence of their presence. These errors include key problems for C and C++ programs: buffer overflows, dangling-pointer errors, and memory leaks. Evidence-based dynamic analysis is fast because it lets the application run at full speed until an error is detected; execution is then rolled back and replayed with instrumentation at the point where the evidence was found, pinpointing the error. We present DOUBLETAKE, the first evidence-based dynamic analysis framework, and implement these analyses inside it. The resulting analyses are the fastest versions to date, demonstrating the effectiveness and efficiency of this new dynamic analysis approach.

References

[1] P. Akritidis, M. Costa, M. Castro, and S. Hand. 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, pages 51–66, Berkeley, CA, USA, 2009. USENIX Association.

[2] G. Altekar and I. Stoica. ODR: Output-deterministic replay for multicore debugging. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 193–206, New York, NY, USA, 2009. ACM.

[3] H. Ayguen and M. Eddington. DUMA - Detect Unintended Memory Access. http://duma.sourceforge.net/.

[4] E. D. Berger, B. G. Zorn, and K. S. McKinley. Composing high-performance memory allocators. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, PLDI '01, pages 114–124, New York, NY, USA, 2001. ACM.

[5] M. D. Bond and K. S. McKinley. Bell: Bit-encoding online memory leak detection. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 61–72, New York, NY, USA, 2006. ACM.

[6] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 1–11, New York, NY, USA, 1995. ACM.

[7] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '03, pages 265–275, Washington, DC, USA, 2003. IEEE Computer Society.

[8] D. Bruening and Q. Zhao. Practical memory checking with dr. memory. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 213–223, Washington, DC, USA, 2011. IEEE Computer Society.

[9] J. Caballero, G. Grieco, M. Marron, and A. Nappa. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 133–143, New York, NY, USA, 2012. ACM.

[10] L. Chew and D. Lie. Kivati: fast detection and prevention of atomicity violations. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 307–320, New York, NY, USA, 2010. ACM.

[11] T.-c. Chiueh. Fast bounds checking using debug register. In *Proceedings of the 3rd international conference on High performance embedded architectures and compilers*, HiPEAC'08, pages 99–113, Berlin, Heidelberg, 2008. Springer-Verlag.

[12] J. Chow, T. Garfinkel, and P. M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ATC'08, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.

[13] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *In Proceedings of the 7th USENIX Security Symposium*, pages 63–78, 1998.

[14] H. Cui, J. Wu, C.-C. Tsai, and J. Yang. Stable deterministic multithreading through schedule memoization. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–13, Berkeley, CA, USA, 2010. USENIX Association.

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

[16] N. Hasabnis, A. Misra, and R. Sekar. Light-weight bounds checking. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 135–144, New York, NY, USA, 2012. ACM.

[17] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *In Proc. of the Winter 1992 USENIX Conference*, pages 125–138, 1991.

[18] M. Hauswirth and T. M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, pages 156–164, New York, NY, USA, 2004. ACM.

[19] T. Hoger. "vim: heap buffer overflow". https://bugzilla.redhat.com/show_bug.cgi?id=455455.

[20] Intel Corporation. Intel inspector xe 2013. http://software.intel.com/en-us/intel-inspector-xe, 2012.

[21] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 1–1, Berkeley, CA, USA, 2005. USENIX Association.

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

[23] D. Lea. The GNU C library. http://www.gnu.org/software/libc/libc.html.

[24] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn. 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, pages 77–90, New York, NY, USA, 2010. ACM.

[25] M. O. X. D. Library. Enabling the malloc debugging features. https://developer.apple.com/library/mac/#documentation/performance/Conceptual/ManagingMemory/Articles/MallocDebug.html.

[26] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou. Bugbench: Benchmarks for evaluating bug detection tools. In *In Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[27] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[28] Microfocus. Micro focus devpartner boundschecker. http://www.microfocus.com/store/devpartner/boundschecker.aspx, 2011.

[29] G. C. N. Necula, M. Scott, and W. Westley. Ccured: Type-safe retrofitting of legacy code. In *Proceedings of the Principles of Programming Languages*, pages 128–139, 2002.

[30] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[31] E. B. Nightingale, D. Peek, P. M. Chen, and J. Flinn. Parallelizing security checks on commodity hardware. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 308–318, New York, NY, USA, 2008. ACM.

[32] G. Novark, E. D. Berger, and B. G. Zorn. Exterminator: automatically correcting memory errors with high probability. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1–11, New York, NY, USA, 2007. ACM Press.

[33] G. Novark, E. D. Berger, and B. G. Zorn. Efficiently and precisely locating memory leaks and bloat. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 397–407, New York, NY, USA, 2009. ACM.

[34] Oracle Corporation. Sun memory error discovery tool (discover). http://docs.oracle.com/cd/E18659_01/html/821-1784/gentextid-302.html.

[35] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. PRES: probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 177–192, New York, NY, USA, 2009. ACM.

[36] B. Perens. Electric Fence. http://perens.com/FreeSoftware/ElectricFence/.

[37] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. 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, pages 235–248, New York, NY, USA, 2005. ACM.

[38] M. Ronsse and K. De Bosschere. RecPlay: a fully integrated practical record/replay system. *ACM Trans. Comput. Syst.*, 17(2):133–152, May 1999.

[39] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: a fast address sanity checker. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, USENIX ATC'12, pages 28–28, Berkeley, CA, USA, 2012. USENIX Association.

[40] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: a lightweight extension for rollback and deterministic replay for software debugging. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '04, pages 3–3, Berkeley, CA, USA, 2004. USENIX Association.

[41] J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou. Triage: Diagnosing production run failures at the user's site. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 131–144, New York, NY, USA, 2007. ACM.

[42] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. DoublePlay: parallelizing sequential logging and replay. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS XVI, pages 15–26, New York, NY, USA, 2011. ACM.

[43] A. Welc, B. Saha, and A.-R. Adl-Tabatabai. Irrevocable transactions and their applications. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, SPAA '08, pages 285–296, New York, NY, USA, 2008. ACM.

[44] P. R. Wilson. Uniprocessor garbage collection techniques. In *Proceedings of the International Workshop on Memory Management*, IWMM '92, pages 1–42, London, UK, UK, 1992. Springer-Verlag.

[45] Q. Zeng, D. Wu, and P. Liu. 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, pages 367–377, New York, NY, USA, 2011. ACM.