DOUBLETAKE: Fast and Precise Error Detection via Evidence-Based Dynamic Analysis

Tongping Liu *
Dept. of Computer Science
University of Texas
at San Antonio
San Antonio, TX 78249
Tongping.Liu@utsa.edu

Charlie Curtsinger *
Dept. of Computer Science
Grinnell College
1116 8th Ave.
Grinnell, IA 50112
curtsinger@cs.grinnell.edu

Emery D. Berger
College of Information and
Computer Sciences
University of Massachusetts
Amherst
Amherst, MA 01003
emery@cs.umass.edu

ABSTRACT

Programs written in unsafe languages like C and C++ often suffer from errors like buffer overflows, dangling pointers, and memory leaks. Dynamic analysis tools like Valgrind can detect these errors, but their overhead—primarily due to the cost of instrumenting every memory read and write—makes them too heavyweight for use in deployed applications and makes testing with them painfully slow. The result is that much deployed software remains susceptible to these bugs, which are notoriously difficult to track down.

This paper presents *evidence-based dynamic analysis*, an approach that enables lightweight analyses—under 5% overhead for these bugs—making it practical for the first time to perform these analyses in deployed settings. The key insight of evidence-based dynamic analysis is that for a class of errors, it is possible to ensure that evidence that they happened at some point in the past remains for later detection. Evidence-based dynamic analysis allows execution to proceed at nearly full speed until the end of an epoch (e.g., a heavyweight system call). It then examines program state to check for evidence that an error occurred at some time during that epoch. If so, it rolls back execution and re-executes the code with instrumentation activated to pinpoint the error.

We present DOUBLETAKE, a prototype evidence-based dynamic analysis framework. DOUBLETAKE is practical and easy to deploy, requiring neither custom hardware, compiler, nor operating system support. We demonstrate DOUBLETAKE's generality and efficiency by building dynamic analyses that find buffer overflows, memory use-after-free errors, and memory leaks. Our evaluation shows that DOUBLETAKE is efficient, imposing just 4% overhead on average, making it the fastest such system to date. It is also precise: DOUBLETAKE pinpoints the location of these errors to the exact line and memory addresses where they occur, providing valuable debugging information to programmers.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging–Debugging Aids, Monitors, Tracing; D.2.4 [**Software Engineering**]: Software/Program Verification–Reliability; D.3.4 [**Programming Languages**]: Run-time environments

General Terms

Performance, Reliability

Keywords

Dynamic Analysis, Software Quality, Testing, Debugging, Leak Detection, Buffer Overflow Detection, Use-After-Free Detection

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 [28]. Valgrind's most popular use case, via its default tool, MemCheck, can find a wide range of memory errors, including buffer overflows, use-after-free errors, and memory leaks.

Unfortunately, these dynamic analysis tools often impose significant performance overheads that make them prohibitive for use outside of testing scenarios. An extreme example is the widely-used tool Valgrind. Across the SPEC CPU2006 benchmark suite, Valgrind degrades performance by almost 17× on average (geometric mean); its overhead ranges from 4.5× and 42.8×, making it often too slow to use even for testing (see Table 1).

While faster dynamic analysis frameworks exist for finding particular errors (leveraging compiler support to reduce overhead), they sacrifice precision while continuing to impose substantial overhead that would impede their use in deployed settings. The current state-of-the-art, Google's AddressSanitizer, detects buffer overflows and use-after-free errors, but slows applications by around 30% [38]. AddressSanitizer also identifies memory leaks but only at the end of program execution, which is not useful for servers or other long-lived applications.

Because of their overhead, this class of dynamic analysis tools can generally only be used during testing. However, they are limited by definition to the executions that are tested prior to deployment. Even exhaustive testing regimes will inevitably fail to uncover these errors, which are notoriously difficult to debug.

This paper presents an approach called *evidence-based dynamic analysis* that is based on the following key insight: it is often possible to discover evidence that an error occurred or plant markers that ensure that such evidence exists. By combining evidence placement with checkpointing and infrequent checking, we can run applications at nearly full speed in the common case (no errors). If we find an error, we can use the checkpoint to roll back and re-execute the program with instrumentation activated to pinpoint the exact cause of the error.

Certain errors, including the ones we describe here, naturally exhibit a *monotonicity* property: when an error occurs, evidence that it

---

*This work was initiated and partially conducted while Liu and Curtsinger were PhD students at the University of Massachusetts Amherst.

happened tends to remain or even grow so that it can be discovered at a later point during execution. When this evidence is not naturally occurring or not naturally monotonic, it can be forced to exhibit this property by *planting* evidence via what we call *tripwires* to ensure later detection. A canonical example of such a tripwire is a random value, also known as a *canary*, placed in unallocated space between heap objects [11]. A corrupted canary is incontrovertible evidence that a buffer overflow occurred at some time in the past.

This paper presents a prototype evidence-based dynamic analysis framework called DOUBLETAKE that locates such errors with extremely low overhead and no false positives. DOUBLETAKE checkpoints program state and performs most of its error analyses only at epoch boundaries (what we call irrevocable system calls) or when segfaults occur; these occur relatively infrequently, amortizing DOUBLETAKE's overhead.

If DOUBLETAKE finds evidence of an error at an epoch boundary or after a segmentation violation, it re-executes the application from the most recent checkpoint. During re-execution, DOUBLETAKE enables instrumentation to let it precisely locate the source of the error. For example, for buffer overflows, DOUBLETAKE sets hardware watchpoints on the tripwire memory locations that were found to be corrupted. During re-execution, DOUBLETAKE pinpoints exactly the point where the buffer overflow occurred.

We have implemented DOUBLETAKE as a drop-in library that can be linked directly with the application, without the need to modify code or even recompile the program. DOUBLETAKE works without the need for custom hardware, compiler, or OS support.

Using DOUBLETAKE as a framework, we have built three different analyses that attack three of the most salient problems for unsafe code: the buffer overflow detector described above as well as a use-after-free detector and memory leak detector. These analyses can all run concurrently. By virtue of being evidence-based, they have a zero false positive rate, precisely pinpoint the error location, and operate with *extremely* low overhead: for example, with DOUBLETAKE, buffer overflow analysis alone operates with just 3% overhead on average. When all three of these analyses are enabled, DOUBLETAKE's average overhead is under 5%.

For all of the analyses we have implemented, DOUBLETAKE is the fastest detector of these errors to date, providing compelling evidence for the promise of evidence-based dynamic analyses. Its overhead is already low enough to dramatically speed testing and often low enough to enable the use of these formerly-prohibitive analyses in deployed settings. This work thus promises to significantly extend the reach of dynamic analyses.

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 prototype framework that implements evidence-based dynamic analyses for C/C++ programs: each of the analyses we have built using DOUBLETAKE – detecting buffer overflows, use-after-frees, and memory leaks – are the fastest reported to date.

Outline

This paper first provides an overview of the basic operation of DOUBLETAKE in Section 2. Section 3 details the dynamic analyses we have built using DOUBLETAKE. Section 4 describes key implementation details. Section 5 evaluates DOUBLETAKE's effectiveness, performance, and memory overhead, and compares these to the state of the art. Section 6 discusses limitations of evidence-based analysis and the detectors we implement. Section 7 describes key related work and Section 8 concludes.

Valgrind Execution Time Overhead

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 1: Valgrind's execution time overhead across the SPEC benchmark suite. Valgrind imposes on average 17× overhead (geometric mean), making it prohibitively high for use in deployment and quite expensive even for testing purposes.

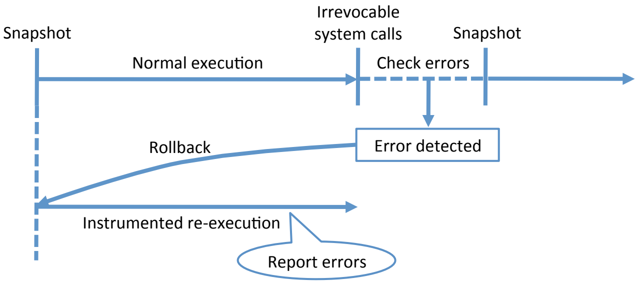


Figure 1: Overview of DOUBLETAKE in action: execution is divided into epochs at the boundary of irrevocable system calls. Each epoch begins by taking a snapshot of program state. Execution runs at nearly full-speed during epochs. Evidence-based analysis takes place once an epoch ends, replaying execution from the previous snapshot until it pinpoints the exact location where the error is introduced. Relatively-long epochs amortize the cost of snapshots and analysis, keeping overhead low.

2. OVERVIEW

DOUBLETAKE is an efficient dynamic analysis framework for a class of errors that exhibit or can be forced to exhibit a *monotonicity* property: evidence of the error is persistent and can be gathered after-the-fact. With DOUBLETAKE, program execution is divided into epochs, during which execution proceeds at full speed (Figure 1). At the beginning of an epoch, DOUBLETAKE checkpoints program state. Epochs end only when the application issues an *irrevocable* system call (e.g., a socket read); most system calls are not irrevocable (see 4.3 for full details). Once an epoch ends, DOUBLETAKE checks the program state for evidence of memory errors. Because epochs are relatively long-lived, the cost of checkpointing and error analysis is amortized over program execution. If DOUBLETAKE finds an error, it re-executes code executed from the previous epoch with additional instrumentation to pinpoint the exact cause of the error.

To demonstrate DOUBLETAKE's effectiveness, we have implemented detection tools for three of the most important classes of errors in C and C++ code: heap buffer overflows, use-after-free errors, and memory leaks (Section 3 describes these in detail). 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. An 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 a small amount of system state at the beginning of each epoch (e.g., file offsets), which lets it unroll the effect of system calls that modify this state when re-execution is required.

During execution, DoubleTake manages various types of system calls in an effort to reduce the number of epochs, which Section 4.3 discusses. In practice, DoubleTake limits the number of epoch boundaries, amortizing the cost of program state checks. The kind of checks employed depend on the particular dynamic analysis being performed; Section 3 describes the details of the analyses we have built on top of DoubleTake.

2.2 Lightweight Replay

When program state checks indicate that an error occurred during the current epoch, DoubleTake replays execution from the last epoch to pinpoint the error's root cause. DoubleTake ensures that all program-visible state, including system call results and memory allocations and deallocations, is identical to the original run. During replay, DoubleTake returns cached return values for most system calls, with special handling for some cases. Section 4 describes in detail how DoubleTake records and re-executes system calls.

2.3 Deterministic Memory Management and Tripwire Support

One key challenge to using replay to find the exact location of errors is that we cannot rely on the default system-supplied heap allocator. The reason for this is that it does not provide a replayable sequence of addresses. The default heap grows on demand by invoking `mmap` (or a similar call on other operating systems) to obtain memory from the system. However, because of address-space layout randomization, now implemented on all modern operating systems to increase security, `mmap` almost always returns different addresses when invoked. This effect means that heap addresses in a replayed execution would likely differ from the original.

DoubleTake therefore replaces the default heap allocator with a heap built with the Heap Layers framework [4]. In addition to providing repeatable sequences of addresses, DoubleTake's heap provides a number of other useful features that improve DoubleTake's efficiency and simplify building analyses using it:

- **Efficiency via large chunk allocation.** The DoubleTake heap obtains memory from the operating system in large chunks and satisfies all memory allocations from it, reducing the number of system calls that DoubleTake must track and thus lowering its overhead.
- **Simplified tripwire installation.** DoubleTake's heap also makes the process of implanting tripwires easier. For example, detection tools can easily interpose on heap operations to alter memory allocation requests or defer the reuse of freed memory, and can mark the status of each object in metadata (e.g., via a dedicated object header that the heap provides for this purpose).
- **Efficient tripwire checking.** Finally, DoubleTake's heap makes tripwire checking far more efficient. It maintains a shadow bitmap to identify the locations and status of heap canaries, which allows it to use vectorized bit operations to perform efficient checking at the end of each epoch.

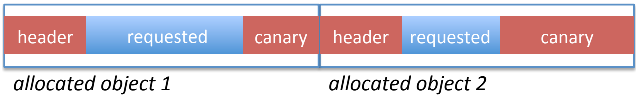


Figure 2: Heap organization used to provide evidence of buffer overflow errors. Object headers and unrequested space within allocated objects are filled with canaries; a corrupted canary indicates an overflow occurred.

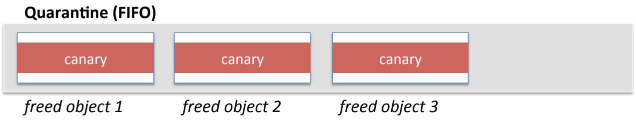


Figure 3: Evidence-based detection of dangling pointer (use-after-free) errors. Freed objects are deferred in a quarantine in FIFO order and filled with canaries. A corrupted canary indicates that a write was performed after an object was freed.

Section 4.3 presents full details of DoubleTake's heap implementation.

2.4 Pinpointing Error Locations

During replay, DoubleTake lets detection tools set *hardware watchpoints* during re-execution to pinpoint error locations (i.e., on an overwritten canary). Modern architectures make available a small number of watchpoints (four on x86). Each watchpoint can be configured to pause program execution when a specific byte or word of memory is accessed. While watchpoints are primarily used by debuggers, DoubleTake uses them to speed error location during re-execution.

DoubleTake's watchpoints are particularly useful in combination with heap canaries. For example, during re-execution, DoubleTake's 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. ANALYSES

To demonstrate DoubleTake's generality and efficiency, we implement a range of error-detection tools as evidence-based dynamic analyses. In particular, we implement the following three detection tools with DoubleTake:

- **Heap buffer overflow detection** (§3.1): when an application writes outside the bounds of an allocated object,
- **Use-after-free detection** (§3.2): when an application writes to freed memory (i.e., through a *dangling pointer*), and
- **Memory leak detection** (§3.3): when a heap object becomes inaccessible but has not been explicitly freed.

For each of these tools, we describe the evidence that DoubleTake observes or places to detect these errors, and how re-execution and error isolation proceeds once an error is detected. Note that because these analyses are orthogonal, they can all be used simultaneously.

3.1 Heap Buffer Overflow Detection

Heap buffer overflows occur when programs write outside the bounds of an allocated object. DOUBLETAKE reports an error when it discovers that a canary value has been overwritten. When it finds an overwritten canary, the detector places watchpoints during re-execution to identify the instruction responsible for the overflow.

Evidence-Based Error Detection

Figure 2 presents an overview of the approach used to locate buffer overflows. Our buffer overflow detector places canaries between heap objects so that an overflow from one object into an adjacent one can be detected.

In addition, the overflow detector fills any remaining empty space inside allocated objects with canaries; DOUBLETAKE's allocator rounds all object size requests up to the nearest power of two. This approach lets DOUBLETAKE identify small overflows that would otherwise be missed because they did not actually go beyond the object's allocated space.

At memory deallocation time (calls to `free` or `delete`), DOUBLETAKE checks for buffer overflows in objects whose requested size is less than a power of two. It defers the checking of power-of-two sized objects to the end of the current epoch.

At the end of each epoch, DOUBLETAKE checks whether any canaries have been overwritten (including those for exact power-of-two requests). If it finds any overwritten canaries, it has incontrovertible evidence that a buffer overflow has occurred. DOUBLETAKE then triggers a re-execution to locate the exact point in the program when the overflow happened.

Re-Execution and Error Isolation

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 complete call stack of the trapped instruction by invoking the `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.

Evidence-Based Error Detection

Figure 3 illustrates how we detect use-after-free errors using DOUBLETAKE. Our use-after-free detector delays the re-allocation of freed memory. We adopt the approach used by AddressSanitizer of maintaining a FIFO quarantine list [38]. In our implementation, objects are released from the quarantine list when the total size of quarantined objects exceeds 16 megabytes, or when there are more than 1,024 quarantined objects. (Note that all thresholds used by the detector are easily configurable.)

The detector overwrites the first 128 bytes of all objects in the quarantine list (which have all been freed by the program) with canary values. This threshold strikes a compromise between error detection and efficiency. We have found empirically that filling larger objects with canaries (i.e., going beyond 128 bytes to the full size of allocated objects) introduces substantial overhead during normal execution, but is unlikely to catch any additional errors. This is because large objects often consist of a header followed by a buffer. A prematurely reused object is likely to have its prologue scrambled by a constructor, while the remainder of the object (the buffer contents) may remain unmodified for a long time.

Before an object can be returned to the program heap, DOUBLETAKE verifies that no canaries have been overwritten. It also checks all canaries in the entire heap at epoch boundaries. In either case, if a canary has been overwritten, the detector knows that a use-after-free error has occurred. It then immediately triggers re-execution to identify the cause of this error.

Re-Execution and Error Isolation

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. The detector also 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. When this signal is triggered, the detector reports information about the object's allocation and deallocation sites, as well as call stack and line number information for the instructions responsible for the use-after-free error.

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 an 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 this information for all`malloc` calls in normal execution would unnecessarily slow down the program 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.

Evidence-Based Error Detection

Unlike the previously-described detectors, memory leak detection does not need tripwires. Instead, the evidence of a memory leak is latent in the heap organization itself.

Our detector finds memory leaks using the same marking approach as conservative garbage collection [42]. 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 each object. Each object has a header containing a *marked* bit and an *allocated* bit. If the *marked* bit is set, this object has already been visited. The detector then removes this object and moves on to the next item in the queue. If the object is allocated but not yet marked, the detector marks it as reachable by setting the *marked* bit and adds all pointer values within the object's bounds to the work queue. Once the work queue is empty, DOUBLETAKE ends its scan.

DOUBLETAKE then traverses the entire heap to find any leaked objects: these are allocated but unmarked (unreachable). If it finds memory leaks, re-execution begins. Note that using this approach, our detector can also find potential dangling pointers (that is, reachable freed objects). This option is disabled by default because, unlike other applications, potential dangling pointer detection could produce false positives.

Re-Execution and Error Isolation

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 since the last site is responsible for the memory leak.

4. IMPLEMENTATION DETAILS

DOUBLETAKE is implemented as a library for Linux applications. It can be linked directly or at runtime using the `LD_PRELOAD` mechanism. DOUBLETAKE is thus convenient to use: there is no need to change or recompile applications, to use a specialized hardware platform, run inside a virtual machine, or modify the OS.

At startup, DOUBLETAKE begins the first epoch. This epoch continues until the program issues an *irrevocable* system call (see Section 4.3 for details). Before an irrevocable system call, DOUBLETAKE checks program state for evidence of errors. The details are presented in Section 3.

If no errors are found, DOUBLETAKE ends the current epoch, issues the irrevocable system call, and begins a new epoch. If it finds evidence of an error, DOUBLETAKE enters re-execution mode. DOUBLETAKE will then re-execute with instrumentation activated and report the lines of code responsible for the error(s).

The remainder of this section describes the implementation of DOUBLETAKE's core functionality.

4.1 Startup and Shutdown

At program startup, DOUBLETAKE performs initialization and starts the first epoch. DOUBLETAKE needs to get in early to interpose on system calls and install its own heap implementation. It accomplishes this by marking its own initialization function with the constructor attribute. Since DOUBLETAKE must wrap library functions that eventually invoke with system calls, as described in Section 4.3, it collects the addresses of all intercepted functions during this initialization phase. DOUBLETAKE acquires memory from the OS to hold its heap, collects the names and ranges of all globals by analyzing `/proc/self/maps`, installs signal handler for segmentation violations, and prepares the data structure for recording and handling system calls.

For technical reasons, DOUBLETAKE must postpone the checkpointing of program state (and thus the beginning of the first epoch) until just before execution enters the application enters its `main` function. This delay is necessary to let key low-level startup tasks complete. For example, C++ performs its initialization for the standard stream after the execution of constructor functions (including, in this case, DOUBLETAKE itself). Because DOUBLETAKE relies on streams to report any errors it detects, by definition it cannot start the first epoch before that point. To make this all possible, we interpose on the `libc_start_main` function, and pass a custom `main` function implemented by DOUBLETAKE that performs a snapshot just before entering the application's real `main` routine.

DOUBLETAKE treats program termination as the end of the final epoch. As with any other epoch, if it finds evidence of program errors, DOUBLETAKE re-executes the program to pinpoint the exact causes of errors. This logic is embedded in a finalizer marked with the deconstructor attribute that DOUBLETAKE installs.

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 2: System calls handled by DOUBLETAKE. All unlisted system calls are conservatively treated as irrevocable, and will end the current epoch. Section 4.3 describes how DOUBLETAKE handles calls in each category.

4.2 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 the register state of each thread with the `getcontext` function. To reduce the cost of snapshots, DOUBLETAKE does not checkpoint any read-only memory. To identify all writable mapped memory, DOUBLETAKE processes the `/proc/self/map` file, which on Linux identifies every mapped memory region and its attributes (other operating systems implement similar functionality). DOUBLETAKE also records the file positions of all open files, which lets programs issue `read` and `write` system calls without ending the current epoch. DOUBLETAKE uses the combination of saved memory state, file positions and registers to rollback execution if it finds evidence of an error.

4.3 Normal Execution

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

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 to enable re-execution.

DOUBLETAKE divides system calls into five categories, shown in Table 2. 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.

- **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 OS.

- **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 operations fall into this category. For example, although `write` modifies file contents, DOUBLETAKE can write the same content during re-execution. The `write` function also changes the current file position, but the file position can be restored to the saved one using `lseek` prior to re-execution.

At the beginning of each epoch, DOUBLETAKE saves all file descriptors of opened files in a hash table. Maintaining this hash table helps to identify whether a `read` and `write` call is operating on sockets or not, because socket communications must be treated as irrevocable system calls. In addition, DOUBLETAKE must save stream contents returned by `fread` in order to support re-execution.

- **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 starting a new epoch when there is no need to re-execute the program.
- **Irrevocable system calls** change internally-visible program state, and cannot be rolled back and re-executed. DOUBLETAKE ends the current epoch before these system calls.

DOUBLETAKE reduces the number of irrevocable system calls by observing their arguments; in some cases, they are not necessarily irrevocable. For example, when `fcntl` invoked with `F_GET`, DOUBLETAKE treats it as a *repeatable system call* since it is simply a read of file system state. However, it treats this call as irrevocable if invoked with `F_SET`, since the call then actually updates the file system.

Memory Management

As described in Section 2.3, DOUBLETAKE intercepts memory allocations and deallocations to implant *tripwires*, identify heap corruption, and 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. We built this heap using the HEAP LAYERS framework [4].

DOUBLETAKE implants tripwires differently for different analyses. To detect heap-based buffer overflows, DOUBLETAKE places canaries along with each heap object. In order to find use-after-free errors, DOUBLETAKE postpones the reuse of freed objects by putting them into a quarantine list and filling them with canaries. For memory leak detection, there is no need to implant tripwires, because the evidence of a leak can be found without them.

To identify heap corruption, DOUBLETAKE maintains a bitmap that records the locations of all heap canaries. The bitmap records every word of heap memory that contains a canary, which will be checked at the end of each epoch. If any of these words are modified, DOUBLETAKE notifies the detection tool.

To speed re-execution, DOUBLETAKE uses its heap allocator to satisfy memory requests from the application and corresponding libraries, and maintains a separate heap for internal use only. For example, the memory that DOUBLETAKE uses to record system calls results is allocated from its internal heap and there is no need to replay these allocations during re-execution. Any additional memory allocations during the replay phase are also satisfied from its internal heap.

4.4 Epoch End

Each epoch ends when the program issues an irrevocable system call. At the end of each epoch, DOUBLETAKE checks program state for errors. These analysis-specific error checks are described in Section 3. If an error is found, DOUBLETAKE rolls back execution to the immediately-preceding epoch, and switches to re-execution mode. If no error is found, DOUBLETAKE issues any deferred system calls, clears the logs for all recorded system calls, and begins the next epoch.

4.5 Rollback

If an error is found, DOUBLETAKE rolls back program state prior to beginning re-execution. This rollback must be handled with care.

For example, restoring the saved stack may corrupt the current stack if the size of the saved stack is larger than that of the current stack. DOUBLETAKE thus switches to a temporary stack during its rollback phase. When performing rollback, the saved state of all writable memory is copied back, which also recovers the status of its heap. DOUBLETAKE also recovers the file positions of opened files so that all read/write calls can be issued normally during re-execution.

DOUBLETAKE then sets hardware watchpoints on all corrupted addresses in order to report the root causes of buffer overflows or dangling pointers. Since debug registers are not directly accessible in user mode, DOUBLETAKE utilizes the `perf_event_open` call to load watched addresses into the debug registers. DOUBLETAKE also sets a `SIGTRAP` handler for watchpoints so that it will get notified when these addresses are overwritten (e.g., during buffer overflows or uses of freed objects).

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

4.6 Re-Execution

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

DOUBLETAKE's heap design and its rollback has recovered the memory state to the snapshotted state. To repeat the replayable memory uses, DOUBLETAKE simply repeats memory allocations and deallocations from applications and libraries according to the program order. The additional memory uses happened in the replay phase, such as bookkeeping the call stack of memory uses, will be satisfied from DOUBLETAKE's internal heap and will not affect the memory uses of applications.

During replay, DOUBLETAKE enables tracking of precise information in the memory allocator: all allocations and deallocations record their calling context so these can be reported later, if needed. Note that recording call sites during ordinary execution would be prohibitively expensive, imposing 20–30% overhead; DOUBLETAKE's strategy removes this overhead from normal execution.

Finally, DOUBLETAKE handles traps caused by accesses to watchpoints. Inside the trap handler, DOUBLETAKE first determines which watchpoint caused the current trap if there are multiple watchpoints. It also filters out any accesses from DOUBLETAKE itself. DOUBLETAKE prints the callsite stack of the instruction responsible for a buffer overflow or use-after-free errors and their memory allocation (or deallocation) sites. For memory leaks, DOUBLETAKE reports the allocation callsite of the leaked object.

5. EVALUATION

We evaluate DOUBLETAKE to demonstrate its efficiency, in terms of execution time, memory overhead, and effectiveness at detecting errors. All experiments are performed on a quiescent Intel Core 2 dual-processor system with 16GB of RAM, running on Linux 3.13.0-53-generic with `glibc-2.19`. Each processor is a 4-core 64-bit Intel Xeon, operating at 2.33GHz with a 4MB shared L2 cache and a 32KB per-core L1 cache. All programs are built as 64-

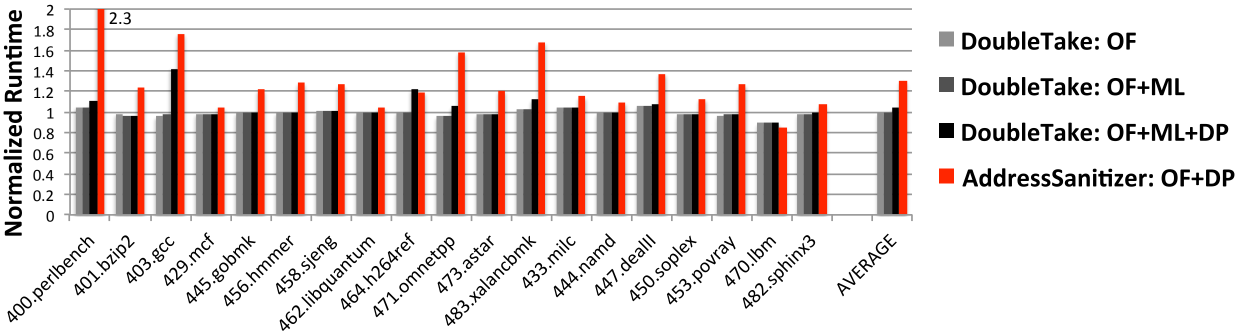


Figure 4: Runtime overhead of DoubleTake (OF = Buffer Overflow Detection, ML = Memory Leak Detection, DP = Dangling Pointers Detection) and AddressSanitizer, normalized to each benchmark's original execution time. With all detections enabled, DoubleTake only introduces 4% performance overhead on average.

bit executables using LLVM 3.2 with the clang front-end and `-O2` optimizations. All evaluations on SPEC CPU2006 are exercised with the "ref" (reference) input set.

5.1 Runtime Overhead

We evaluate DoubleTake's runtime and memory overhead across all of the C and C++ SPEC CPU2006 benchmarks, 19 in total. We compare DoubleTake with the previous state-of-the-art tool, Google's AddressSanitizer [38]. As mentioned earlier, AddressSanitizer can detect buffer overflows and use-after-free errors, but it only detects memory leaks at the end of execution. By contrast, DoubleTake detects all of these errors at the end of every epoch.

In our evaluation, DoubleTake discovered several memory leaks, which trigger rollback and error identification. To isolate normal execution overhead, we disable DoubleTake's rollback in our evaluation. That is, our runs with DoubleTake incur all of the overhead of ordinary tracking (including implanting of tripwires and examining state) but do not measure the time to rollback and locate errors; in general, this cost is low and in any event does not affect bug-free execution, which is the common case. For each benchmark, we report the average runtime of three runs.

Figure 4 presents execution time overhead results for DoubleTake and AddressSanitizer. On average, DoubleTake imposes only 4% overhead *with all three error detectors enabled*. When use-after-free detection (DP) is disabled, DoubleTake exhibits no observable overhead. AddressSanitizer has an average runtime overhead over 30%; recall that AddressSanitizer only performs leak detection at the end of program execution, while DoubleTake performs it every epoch.

For 17 out of 19 benchmarks, DoubleTake outperforms AddressSanitizer. For 14 benchmarks, DoubleTake's runtime overhead with all detectors enabled is under 3%. Unsurprisingly, both DoubleTake and AddressSanitizer substantially outperform Valgrind on all benchmarks.

Four of the benchmarks have higher than average overhead for DoubleTake and AddressSanitizer (`400.perlbench`, `403.gcc`, `464.h264ref`, and `483.xalancbmk`). Both DoubleTake and AddressSanitizer substantially increase these applications' memory footprints (see Table 3). We attribute their increased execution time to this increased memory footprint and its corresponding increased cache and TLB pressure.

DoubleTake's use-after-free detection adds roughly 4% runtime overhead, but only `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. Thus, DoubleTake runs much slower for the application `gcc` when the detection of use-after-free errors is enabled. `h264ref` adds significant overhead on DoubleTake because of its large number of epochs.

Table 5 presents detailed benchmark characteristics. The "Processes" column shows the number of different process invocations (by calling `fork`). The number of epochs is significantly lower than the number of actual system calls, demonstrating DoubleTake's effectiveness at reducing epochs via its lightweight system call handling. The benchmarks with the highest overhead share the following characteristics: they consist of a substantial number of epochs (e.g., `perlbench` and `h264ref`) or are unusually `malloc`-intensive (e.g., `gcc`, `omnetpp`, and `xalancbmk`).

Runtime Overhead Summary: For nearly all of the benchmarks we examine, DoubleTake substantially outperforms the state of the art. For most benchmarks, DoubleTake's runtime overhead is under 3%.

5.2 Memory Overhead

We measure program memory usage by recording the peak *physical* memory usage. Virtual memory consumption is generally not relevant for 64-bit platforms, which have enormous virtual address ranges. DoubleTake's pre-allocated heap and internal heap consume 8GB of virtual memory space. We compute peak physical memory usage by periodically collecting process-level information (on Linux, this is available in the `/proc/self/smaps` pseudo-file), and summing the proportional set sizes of memory segments.

Figure 5 presents memory overhead for DoubleTake and AddressSanitizer (Table 3 has full details). On average, both across the benchmark suite and when broken down by footprint (large (> 100MB) and small (< 100MB)), DoubleTake imposes considerably lower memory overhead than AddressSanitizer. DoubleTake imposes lower memory overhead than AddressSanitizer on all but three benchmarks: `perlbench`, `h264`, and `namd`.

We drill down to explore the application and analysis characteristics that contribute to DoubleTake's memory overhead:

- **Number of epochs:** Much of DoubleTake's memory overhead comes from the snapshot of writable memory taken at the beginning of each epoch. However, the first snapshot is often small because the heap is almost empty before the `main` routine. For example, the benchmarks `bzip2`, `mcf`, `sjeng`, `milc`, and `lbm` run in a single epoch, and accordingly exhibit very low memory overhead.

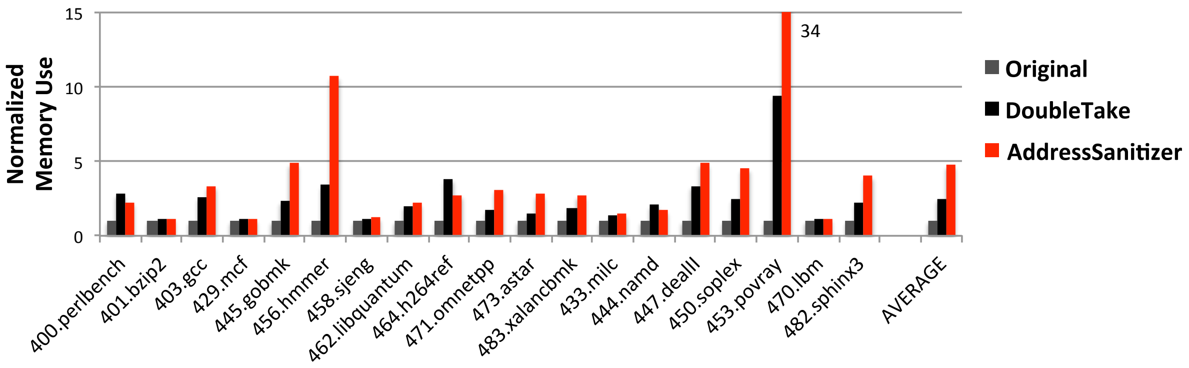


Figure 5: Memory overhead of DoubleTake and AddressSanitizer.

- **System call logs:** System call logs introduce additional memory overhead that depends on the number of recorded system calls.
- **Analysis-specific overhead:** Other sources of memory overhead are analysis-specific. Buffer overflow detection adds canaries between heap objects, which can increase memory usage for programs with many small allocations. For this analysis, DoubleTake also maintains a bit map that marks the placement of canaries: for each eight-bytes word of the heap, DoubleTake adds a bit to mark whether this word has the canaries or not. Finally, use-after-free detection adds constant-size memory overhead by delaying memory reuse. Note that any similar dynamic analyses must impose similar overheads.

Memory Overhead Summary: On average, DoubleTake imposes lower memory overhead than AddressSanitizer. For large footprint applications, it increases memory consumption for applications by 72% on average.

5.3 Effectiveness

We evaluate the effectiveness of DoubleTake on a range of applications, including synthetic test cases, standard benchmarks, and real-world applications.

Synthetic test cases and benchmarks: We first evaluate DoubleTake on 3 synthetic test cases, 26 test cases from the NIST SAMATE Reference Dataset Project. This corpus includes 14 cases with buffer overflows and 12 cases without overflows [17]. We also evaluate DoubleTake on 19 C/C++ benchmarks from the SPEC CPU2006 benchmark suite.

For heap overflows, DoubleTake detects all known overflows in one synthetic test case and 14 test cases of SAMATE suite. For the 12 cases without overflows in SAMATE suite, DoubleTake has no false positives. For the SPEC CPU2006 benchmarks, DoubleTake did not find any heap buffer overflows and use-after-frees, which is the same result found with AddressSanitizer. However, DoubleTake detected a significant number of memory leaks in `perlbench` and `gcc` of SPEC CPU2006, which we verified using Valgrind's Memcheck tool.

Real applications: We also ran DoubleTake with a variety of applications with known errors or implanted errors, listed in Table 4. To verify the effectiveness of DoubleTake's buffer overflow detection, we collected applications from evaluations of prior buffer overflow detection tools, Bugzilla, and bugbench [16, 20, 23, 43], including `bc`, `gcc-4.4.7`, `gzip`, `libHX`, `polymorph`, and `vim-6.3`.

Memory Usage (MB)

Benchmark	Original	Address Sanitizer	DoubleTake
large footprint (> 100MB)			
400.perlbench	656	1481	1834
401.bzip2	870	1020	989
403.gcc	683	2293	1791
429.mcf	1716	1951	2000
458.sjeng	179	220	212
471.omnetpp	172	538	299
473.astar	333	923	479
483.xalancbmk	428	1149	801
433.milc	695	1008	917
447.dealII	514	2496	1724
450.soplex	441	1991	1104
470.lbm	418	496	477
geometric mean		*+117%*	*+72%*
small footprint (< 100MB)			
445.gobmk	28	137	66
456.hmmer	24	256	82
462.libquantum	66	144	132
464.h264ref	65	179	247
444.namd	46	79	96
453.povray	3	133	37
482.sphinx3	45	181	103
geometric mean		*+395%*	*+208%*
overall geometric mean		*+194%*	*+114%*

Table 3: AddressSanitizer and DoubleTake memory usage, in megabytes. The top section lists memory overhead for large-footprint applications (over 100 MB), while the bottom section presents overhead for small-footprint applications. DoubleTake's memory overhead is generally less than AddressSanitizer's.

In every case, DoubleTake detected all known or converted errors. Converted errors are existing global or array overflows that DoubleTake currently cannot detect; we converted these to heap overflows to verify its effectiveness. DoubleTake also identified memory leaks in `gcc-4.4.7` and `vim-6.3`, which we confirmed with Valgrind. To evaluate the detection of use-after-free errors, we manually injected errors on real applications, such as `vim-7.3`, `ls` and `wc`. DoubleTake identified all of these memory errors.

Note that the errors observed in these applications are triggered only by specific inputs. In the common case, these applications perform as expected. This is exactly the case for which DoubleTake

Application	Description	LOC	Error Type
bc	basic calculator	12K	Known Overflow
gzip	compress or expand files	5K	Converted Overflow
libHX	common library	7K	Known Overflow
polymorph	filename converter	0.4K	Converted Overflow
vim-6.3	text editor	282K	Known Overflow
gcc-4.7	GNU Compiler Collection	5784K	Unknown leaks
vim-6.3	text editor	282K	Unknown leak
ls	directory listing	3.5K	Implanted UAF
wc	word count	0.6K	Implanted UAF
vim-7.4	text editor	332K	Implanted UAF

Table 4: Error detection: DOUBLETAKE detects both known (injected and non-injected) and previously unknown errors on the above applications (any reported errors are real, as DOUBLETAKE has a zero false positive rate).

Benchmark	Processes	Epochs	Syscalls	# Mallocs
400.perlbench	3	43	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	825	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 5: Benchmark characteristics.

is ideal, since its low overhead is designed to make it feasible to use it in deployed settings.

Detailed reporting: DOUBLETAKE reports precise information aimed at helping programmers identify the exact causes of different memory errors, as shown in Figure 6(a). For buffer overflows, DOUBLETAKE reports the call sites and line numbers of the overflow and the original memory allocation. For memory leaks, DOUBLETAKE reports the last call site of its memory allocation. For use-after-frees error, DOUBLETAKE reports both allocation and deallocation call sites, and the instruction(s) that wrote to the object after it was freed. In general, DOUBLETAKE provides more detailed information than AddressSanitizer, as seen in Figure 6(b).

In addition, DOUBLETAKE can identify more errors than AddressSanitizer. DOUBLETAKE can track up to four buffer overflows or use-after-free errors during the same epoch because its isolation is based on the use of hardware debugging registers. AddressSanitizer always stops at the detection of the first such error.

Effectiveness Summary: Across the applications we examine, DOUBLETAKE detects all known or injected errors with no false positives. DOUBLETAKE is as effective at finding errors as AddressSanitizer, but with much lower performance and memory overhead. It also provides more detailed reports for these errors.

6. DISCUSSION

The analyses we have built using DOUBLETAKE (heap buffer overflows, use-after-free errors, and memory leaks) have no false positives, but they can have false negatives. Our heap buffer overflow detector cannot identify all non-contiguous buffer overflows, a limitation of as all canary-based detectors. If an overflow touches memory only in adjacent objects and skips over canaries, DOUBLETAKE's end-of-epoch scan will not reveal any evidence of the overflow. Both the buffer overflow and use-after-free detectors can detect errors only on writes. To reduce overhead, the use-after-free detector only places canaries in the first 128 bytes of freed objects. If a write to freed memory goes beyond 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 (a limitation shared by AddressSanitizer). In practice, DOUBLETAKE's epochs are long enough to collect allocation site information for all leaks detected during our evaluation.

While evidence-based dynamic analyses can run with very low overhead, they cannot detect errors if there is no evidence, or it is not practical to force evidence of their existence. Evidence-based analysis also depends on errors being generally monotonic: once an error has occurred, its evidence needs to persist until the end of the epoch in order to ensure detection.

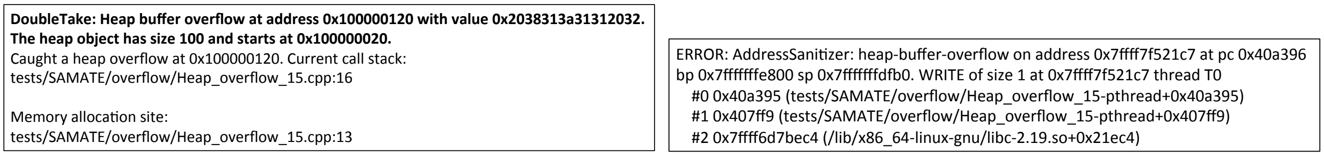
Finally, the current prototype of DOUBLETAKE is limited to executing single-threaded code. However, we believe this is primarily an engineering question. Evidence-based analysis does not depend on fully deterministic replay. Consider the case of a memory error arising due to a race. During replay, the same sequence of writes may not recur, and thus the hardware watchpoints in those addresses might not be triggered. In this scenario, DOUBLETAKE can simply continue execution having successfully masked the error, or repeatedly re-execute the epoch in an effort to expose the data race. Because of this flexibility, DOUBLETAKE will not need to track details about the ordering of memory accesses, which is what makes deterministic record-and-replay systems expensive. Supporting multithreaded programs will only require interception of synchronization operations to allow DOUBLETAKE to pause threads at epoch boundaries and to track the order of synchronization operations.

7. RELATED WORK

Dynamic Instrumentation: Numerous 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, 14, 18, 28, 32]. These tools use dynamic instrumentation engines, such as Pin, Valgrind, and DynamiRIO [7, 24, 28]. 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 ease of use generally 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 5% performance overhead.

Several dynamic analysis tools leverage static analysis to reduce the amount and thus the overhead of instrumentation [1, 12, 13, 27, 33, 38]. While these tools generally reduce overhead over approaches based exclusively on dynamic instrumentation, but cannot detect errors in code that was not recompiled with this instrumentation in place (e.g., inside libraries). In addition, our results show that AddressSanitizer (the previous state-of-the-art, which depends on static analysis) is considerably slower than DOUBLETAKE, which can also perform its analysis on the entire program (including libraries) with no recompilation.

Interposition: DOUBLETAKE uses library interposition exclu-



```
DoubleTake: Heap buffer overflow at address 0x100000120 with value 0x2038313a31312032.
The heap object has size 100 and starts at 0x100000020.
Caught a heap overflow at 0x100000120. Current call stack:
tests/SAMATE/overflow/Heap_overflow_15.cpp:16

Memory allocation site:
tests/SAMATE/overflow/Heap_overflow_15.cpp:13
```

(a) DOUBLETAKE Report

```
ERROR: AddressSanitizer: heap-buffer-overflow on address 0x7ffff7f521c7 at pc 0x40a396
bp 0x7fffffffe800 sp 0x7fffffffdfb0. WRITE of size 1 at 0x7ffff7f521c7 thread T0
   #0 0x40a395 (tests/SAMATE/overflow/Heap_overflow_15-pthread+0x40a395)
   #1 0x407ff9 (tests/SAMATE/overflow/Heap_overflow_15-pthread+0x407ff9)
   #2 0x7ffff6d7bec4 (/lib/x86_64-linux-gnu/libc-2.19.so+0x21ec4)
```

(b) AddressSanitizer Report

Figure 6: Example reports of DOUBLETAKE and AddressSanitizer for buffer overflow identification.

sively 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 [26]. Many prior approaches use a mix of library interposition and virtual memory techniques to detect memory errors [3, 5, 9, 15, 21, 25, 30, 31, 35, 43], though their overhead is much higher than DOUBLETAKE's.

Record and replay: Several replay-based approaches target software debugging and/or fault tolerance [6, 19, 36, 37, 39, 40]. Flashback supports replay debugging by employing a shadow process to checkpoint the state of an application, and recording the results of system calls to facilitate the replay. Triage uses replay to automate the failure diagnosis process for crashing bugs [40]. Both Flashback and Triage need custom kernel support.

Aftersight is the related work that is closest in spirit to DOUBLETAKE [10]. It separates analysis from normal execution by logging inputs to a virtual machine and exporting them to a separate virtual machine for detailed (slow) analysis that can run offline or concurrently with application execution. Aftersight monitors applications running in a virtual machine, which adds some amount of workload-dependent overhead. VM-based recording alone adds additional runtime overhead, an average of 5% on the SPEC CPU2006 benchmarks. Aftersight's dynamic analyses are offloaded to unused processors, which may not be available in some deployments. Unlike Aftersight, DOUBLETAKE does not require the use of a virtual machine, does not rely on additional processors for dynamic analyses, and incurs lower average overhead.

Speck is another replay-based system focused on security checking, including taint analysis and virus scanning [29]. Security checks generally require applications to halt immediately upon detecting an error, functionality that DOUBLETAKE by design does not provide. Other systems have focused on reducing the performance overhead of recording [2, 22, 34, 41].

8. CONCLUSION

This paper introduces *evidence-based dynamic analysis*, a new lightweight dynamic analysis technique. Evidence-based dynamic analysis works for errors that naturally leave evidence of their occurrence, or can be forced to do so. 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, an evidence-based dynamic analysis framework, and implement these analyses using it. The resulting analyses are the fastest to date, imposing on average under 5% overhead. These results demonstrate the effectiveness and efficiency of this approach, which promises to speed testing and dramatically increase the reach of dynamic analysis by extending it to deployed settings. DOUBLETAKE is available for download at `http://github.com/plasma-umass/DoubleTake`.

9. ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation under Grant No. CNS-1525888. Charlie Curtsinger was supported by a Google PhD Research Fellowship. We thank Kostya Serebryany for his assistance with our evaluation of AddressSanitizer, and Qiang Zeng, Dinghao Wu and Peng Liu for providing the test cases used in their Cruiser paper. We also thank Scott Kaplan for his suggestions and comments during the development of DOUBLETAKE. Finally, we thank John Vilk for his invaluable suggestions that helped improve this paper.

10. 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] 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.

[11] 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.

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

[13] 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.

[14] 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.

[15] 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.

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

[17] Information Technology Laboratory, Software and Systems Division. Nist samate reference dataset project. `http://samate.nist.gov/SRD/`, 2013.

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

[19] 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.

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

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

[22] 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.

[23] 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.

[24] 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.

[25] Mac OS X Develper Library. Enabling the malloc debugging features. `https://developer.apple.com/library/mac/#documentation/performance/Conceptual/ManagingMemory/Articles/MallocDebug.html`.

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

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

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

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

[34] 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.

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

[36] 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.

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

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.