Practical Memory Checking with Dr. Memory

Derek Bruening
Google
bruening@google.com

Qin Zhao
Massachusetts Institute of Technology
qin_zhao@csail.mit.edu

***Abstract*—Memory corruption, reading uninitialized memory, using freed memory, and other memory-related errors are among the most difficult programming bugs to identify and fix due to the delay and non-determinism linking the error to an observable symptom. Dedicated *memory checking tools* are invaluable for finding these errors. However, such tools are difficult to build, and because they must monitor all memory accesses by the application, they incur significant overhead. Accuracy is another challenge: memory errors are not always straightforward to identify, and numerous *false positive* error reports can make a tool unusable. A third obstacle to creating such a tool is that it depends on low-level operating system and architectural details, making it difficult to port to other platforms and difficult to target proprietary systems like Windows.**

This paper presents *Dr. Memory*, a memory checking tool that operates on both Windows and Linux applications. Dr. Memory handles the complex and not fully documented Windows environment, and avoids reporting false positive memory leaks that plague traditional leak locating algorithms. Dr. Memory employs efficient instrumentation techniques; a direct comparison with the state-of-the-art Valgrind Memcheck tool reveals that Dr. Memory is twice as fast as Memcheck on average and up to four times faster on individual benchmarks.

***Index Terms*—Memory Checking, Shadow Memory, Dynamic Optimization**

I. Introduction

Memory-related errors are among the most problematic programming bugs. These errors include use of memory after freeing it, reads of uninitialized memory, memory corruption, and memory leaks. Observable symptoms resulting from memory bugs are often delayed and non-deterministic, making these errors difficult to discover during regular testing. Strategies for detecting memory bugs usually rely on randomly encountering visible symptoms during regular application execution. Furthermore, the sources of these bugs are painful and time-consuming to discover from observed crashes that occur much later than the initial memory error. For all of these reasons, memory bugs often remain in shipped products and can show up in customer usage. Dedicated *memory checking tools* are needed to systematically find memory-related errors.

A. Memory Checking Tools

Memory checking tools in use today include Valgrind Memcheck [16], Purify [7], Intel Parallel Inspector [9], and Insure++ [15]. Such tools identify references to freed memory, array bounds overflows, reads of uninitialized memory, invalid calls to heap routines such as double frees, and memory leaks. These tools are large, complex systems that are difficult to build and perfect.

There are three major challenges in building a memory checking tool. Performance is a serious challenge for these tools because of the heavyweight monitoring required to detect memory errors. Application heap and stack allocations must be tracked, and each memory access by the application must be checked by the tool, incurring significant overhead.

Accuracy is another challenge: valid memory operations that appear to violate memory usage rules are not uncommon in system libraries and other locations out of control of the tool user, and *false positives* must be avoided while still detecting real errors. The largest class of false positives are uninitialized read errors stemming from copying whole words containing valid sub-word data (discussed in Section II-A); avoiding these false positives requires monitoring not only memory accesses but nearly every single application instruction, further decreasing performance. Accuracy is also an issue for leak checking. To detect leaks, most tools perform a garbage-collection-style scan at application exit to identify unreachable heap allocations. However, without semantic information assumptions must be made and can lead to false positives.

A third challenge in creating memory checking tools is their dependence on low-level operating system and architectural details. Notably, the most popular non-commercial tool, Memcheck, remains a UNIX-only tool despite widespread demand for a Windows port. Since these tools typically operate in user mode, memory references made by the operating system cannot be directly monitored by the tool. Instead, each system call's effects on application memory must be emulated. This requires detailed knowledge of each parameter to each system call, which is not easy to obtain for proprietary systems like Windows. Additionally, the Windows programming environment contains multiple heap libraries, some relatively complex, which complicate heap monitoring.

This paper presents *Dr. Memory*, a memory checking tool with novel solutions to the challenges listed above. Improving on each of these issues makes for a more practical tool. Dr. Memory is open-source and available at `http://code.google.com/p/drmemory/`.

B. Contributions

This paper's contributions include:

- We describe the design for a complete memory checking tool that supports both Windows and Linux.
- We enumerate possible function wrapping approaches and present a wrapping technique that is transparent and handles mutual recursion, tailcalls, and layered functions.
- We present a novel technique for identifying and delimiting stack usage within heap memory.
- We categorize the sources of false positives in

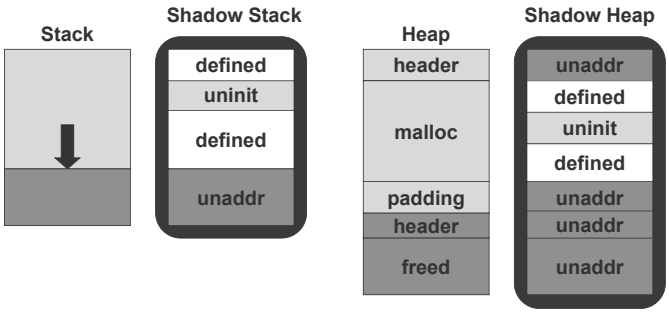

Fig. 1. Example of shadow memory corresponding to application memory within the application stack and heap.

reachability-based leak detection and present novel techniques for avoiding them.
- We present an encoding for callstacks that significantly reduces memory consumption.
- We describe a series of instrumentation techniques and optimizations that improve on the performance of the state-of-the-art memory checking tool, Memcheck, by an average of 2x and by up to 4x on individual benchmarks.

C. Outline

Section II presents the overall design of our memory checking tool and our method of inserting instrumentation into the running application. Section III shows how we achieve good performance with our inserted instrumentation. Next, Section IV discusses wrapping heap library functions, followed by handling kernel interactions in Section V. Section VI describes our leak detection scheme. Experimental results with Dr. Memory are presented in Section VII.

II. System Overview

Dr. Memory uses *memory shadowing* to track properties of a target application's data during execution. Every byte of application memory is shadowed with metadata that indicates one of three states:

- *unaddressable*: memory that is not valid for the application to access.
- *uninitialized*: memory that is addressable but has not been written since it was allocated and should not be read.
- *defined*: memory that is addressable and has been written.

Our notion of addressability is more strict than that provided by the underlying operating system and hardware. In addition to considering invalid memory pages as *unaddressable*, we inspect the interior of the stack and the heap: memory beyond the top of the stack is considered *unaddressable*, and memory on the heap that is outside of an allocated `malloc` block is also *unaddressable*, as shown in Figure 1.

A. Tool Actions

Dr. Memory uses this shadow metadata to identify memory usage errors. Reading or writing unaddressable data is reported as an error. Reading uninitialized data is more complex, however. Sub-word values are typically copied in word units; e.g., a one-byte or two-byte variable will be pushed on the stack or copied elsewhere as a four-byte value (for a 32-bit application). Unless the sub-word variable is explicitly packed into a struct by the programmer, the additional bytes will simply be uninitialized padding bytes. If a tool reports any read of an uninitialized byte as an error, most applications would exhibit numerous *false positives*, making such a tool more difficult to use as true errors are drowned out by false errors. To eliminate these false positives, instead of using shadow metadata that is permanently associated with its corresponding application memory location, we dynamically propagate the shadow values to mirror the application data flow and only report errors on *significant* reads that affect program behavior, such as comparisons for conditional jumps or passing data to a system call. This requires that we shadow not only memory but registers as well. Our implementation propagates shadow values through general-purpose registers but not floating-point or multimedia registers as the latter rarely if ever exhibit the sub-word problem.

Table I summarizes how Dr. Memory maintains the shadow metadata and when it checks for errors. For example, Dr. Memory intercepts calls to library routines that allocate memory, such as `malloc` and `HeapAlloc`, and performs two actions: adjusting the size of the allocation to add *redzones* (see Section IV-A) and updating the shadow memory to indicate that the requested allocation is valid to access. Existing memory checking tools, including Memcheck, follow similar courses of action, though some tools do not propagate shadow values and suffer from the sub-word false positives described above. Propagating shadow metadata requires taking action on nearly every single application instruction. When combining two shadow values during propagation (e.g., two source operands to an arithmetic instruction), the general rule is that *undefined* combined with *defined* results in *undefined*. We encode our shadow values such that we can use a bitwise `or` to compute this operation.

B. Instrumentation System

Dr. Memory is built on the open-source DynamoRIO [4] dynamic instrumentation platform, which provides Dr. Memory with the necessary monitoring capabilities. DynamoRIO uses a *software code cache* to support inserting arbitrary instrumentation that is executed interleaved with copies of the original application instructions (Figure 2).

III. Efficient Implementation

We use a combination of instrumentation techniques to perform the actions in Table I efficiently.

A. Shadow Metadata

Our shadow metadata for registers is stored in directly-addressable thread-local storage, removing any need to spill registers to access it. For memory, we perform a table lookup that translates the application address to its corresponding shadow address. Our translation table divides the address space into identical regions and uses a *shadow block* to store the shadow values for each region.

We use two shadow bits to encode the shadow state of each application byte. An aligned four-byte application word's shadow state can be packed into a single shadow byte and

Category	Application Action	Corresponding Tool Action
library call	`malloc`, `HeapAlloc`	add redzones, mark between as *uninitialized*
library call	`realloc`, `HeapReAlloc`	add redzones, copy old shadow, mark rest as *uninitialized*
library call	`calloc`, `HeapAlloc(HEAP_ZERO_MEMORY)`	add redzones, mark between as *defined*
library call	`free`, `HeapFree`	mark *unaddressable* and delay any re-use by `malloc`
system call	file or anonymous memory map	mark as *defined*
system call	memory unmap	mark as *unaddressable*
system call	pass input parameter to system call	report error if any part of parameter is not *defined*
system call	pass output parameter to system call	report error if any part of parameter is *unaddressable*; if call succeeds, mark memory written by kernel as *defined*
instruction	decrease stack pointer register	mark new portion of stack as *uninitialized*
instruction	increase stack pointer register	mark de-allocated portion of stack as *unaddressable*
instruction	copy from immediate or fp/xmm register	mark target as *defined*
instruction	copy from general-purpose register or memory	copy source shadow to target shadow
instruction	combine 2 sources (arithmetic, logical, etc. operation)	combine source shadows, mirroring application operation, and copy result to target shadow
instruction	access memory via base and/or index register	report error if addressing register is *uninitialized*
instruction	access memory	report error if memory is *unaddressable*
instruction	comparison instruction	report error if any source is *uninitialized*
instruction	copy to floating-point or xmm register	report error if source is *uninitialized*

Table I. The actions taken by Dr. Memory for each application action. All memory starts out in an *unaddressable* state, except the executable and libraries, which are *defined*. The shadow values of memory and registers are propagated in a manner mirroring the corresponding application data flow, with checks for errors at key operations. Redzones are explained in Section IV-A.

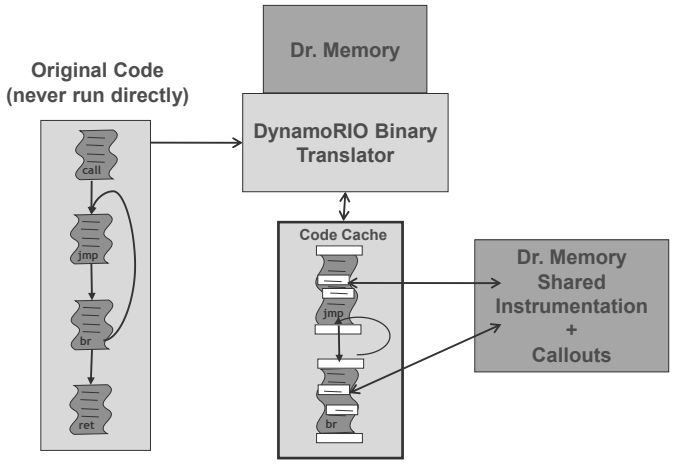

Fig. 2. The components of Dr. Memory. The original application's code is redirected through a software code cache by the DynamoRIO dynamic binary translator. DynamoRIO calls Dr. Memory on each new piece of code, allowing the tool to insert instrumentation around every application instruction.

operated on all at once, to save memory, when races on writing to adjacent bytes are synchronized (at a performance cost) or not a concern (a tradeoff for the user to decide). Most word-sized memory references are word-aligned, and we tune our instrumentation for the aligned case when using packed shadow memory.

B. Fastpath and Slowpath

We divide Dr. Memory's actions into two instrumentation paths: the *fastpath* and the *slowpath*. The fastpath is implemented as a set of carefully hand-crafted machine-code sequences or *kernels* covering the most performance-critical actions. Fastpath kernels are either directly inlined or use shared code with a fast subroutine switch. Obtaining shadow metadata and propagating shadow values for an individual application instruction occurs in inlined instrumentation that is inserted prior to the copy of the application instruction in the code cache. More extensive fastpath kernels like handling stack pointer updates are performed in separate functions with a minimal context switch.

Rarer operations are not worth the effort and extra maintenance costs of using hand-coded kernels and are handled in our slowpath in C code with a full context switch used to call out to the C function. Non-word-aligned memory references, complex instructions, error reporting, and allocation handling are all done in the slowpath. While most of the time execution remains on the fastpath, the slowpath must be efficient as well to avoid becoming a performance bottleneck.

C. Instrumentation Optimizations

We have applied a number of key optimizations to our fastpath kernels, to our general shadow memory handling that is used in both the fastpath and slowpath, and to other aspects of Dr. Memory.

1) Table lookup for addressability: When an aligned four-byte memory reference's shadow value is packed into a single byte, identifying whether any of the 2-bit subsequences is *unaddressable* is best done with a table lookup to cover all possibilities, rather than checking for a few common combinations such as all-*unaddressable* or all-*uninitialized* values and exiting to the slowpath for other combinations, or enumerating all combinations in inlined code.

2) Use faults to detect writes to special shadow blocks: Like Memcheck, Dr. Memory uses unique, shared shadow blocks (*special blocks*) in the translation table to save memory for large address space regions with identical shadow values, such as unallocated space (*unaddressable*) or libraries (*defined*). Rather than adding explicit checks on writes to these

special blocks, Dr. Memory keeps them read-only and uses fault handling to detect when a special block needs to be replaced with a regular block.

3) Whole-basic-block register spilling: Rather than spilling and restoring registers around each application instruction, Dr. Memory analyzes one basic block at a time and shares spill registers across the whole block, reducing the number of spills.

4) Binary callstacks: Printing to a text buffer takes time that can be significant for applications with many allocations. Text callstacks also incur a large memory cost. Section VI-D describes our binary callstack encoding that saves both time and space.

5) Fast replacement routines: Modern string and memory search and copy routines (`strcpy`, `memset`, `strlen`, etc.) are highly optimized in ways that produce false positives in memory checking tools. The typical approach is to replace the application's version of each routine with a simple byte-at-a-time implementation. We found, however, that such an implementation can result in performance degradation, as applications often spend a lot of time in these routines. Using word-at-a-time loops when aligned results in significant performance improvements for Dr. Memory.

6) Sharing shadow translations: Sharing shadow translation results among adjacent application memory references that use identical base and index registers and differ only in displacement avoids redundant shadow table lookups and reduces instrumentation size.

7) Fast stack adjustment handling: Efficient stack adjustment handling is critical for performance of a memory checking tool. Natively, the application performs a simple arithmetic operation on the stack pointer, while the tool must update shadow values for each memory address between the old and new stack pointer. This occurs on every function frame setup and teardown as well as on each push or pop instruction. Dr. Memory uses optimized handlers for specific stack deltas, as well as an optimized general routine that uses a word-at-a-time loop bracketed by byte-at-at-time loops for leading and trailing un-shadow-aligned addresses.

8) Storing displacements in the translation table: Rather than storing the absolute address of each shadow block in the shadow translation table, the displacement from the corresponding application address is stored, reducing the length of the translation code.

These eight optimizations, combined with our fastpath kernels, result in efficient instrumentation. Some of Dr. Memory's most complex operations, however, happen in the slowpath and are discussed next: tracking heap allocations.

IV. Heap Tracking

Many of our heap tracking design decisions were shaped by the complex Windows environment. Windows applications have several heap libraries to choose from. At the lowest level are a set of heap routines in the `ntdll.dll` system library's `Rtl` interface. Above that, the Windows API provides multiple heap region support and control over many aspects of heap allocation, including synchronization and anti-fragmentation strategies. Additionally, C and C++ applications can use the C library's `malloc` and `free` and the C++ operators `new` and `delete`. Implementations of the C library routines ultimately use the Windows API routines to obtain memory, but in some instances they perform their own sub-allocations. The Windows API routines in turn invoke the `Rtl` functions, which use system calls to acquire new memory for parceling out. These multiple layers of heap routines, with possible direct invocations from an application or library to each layer, and with the possibility of sub-allocation at each layer, complicate heap tracking.

A. Wrapping Versus Replacing

A memory checking tool must monitor heap allocations in order to identify errors when reading or writing beyond the bounds of allocated heap memory. Most tools add a *redzone* around each allocation to increase the chance that heap overflows will not be mistaken for an access to an adjacent allocation. Tools also delay frees in an attempt to detect use-after-free errors.

There are two approaches to monitoring and modifying heap library calls: *wrapping* and *replacing*. Wrapping lets the original library code execute but adds a prologue and epilogue where the arguments and return value can be modified and bookkeeping can be updated. Replacing uses a custom function and does not execute any of the original library function code.

The memory checking tool Memcheck [16], which operates on a range of UNIX operating systems, replaces heap library functions. However, on Windows, the additional complexity of the heap API led to our decision to wrap instead of replace. To wrap we only need to understand and depend on a known subset of the API, whereas to replace we would need to emulate every nuance of the entire API, including heap validation, serialization and exception-throwing options, and, the most challenging, undocumented features at the `Rtl` layer such as *heap tags*. Wrapping has other advantages over replacing. Wrapping preserves a heap layout that is more faithful to the native execution of the application, though it may be distorted by redzones. This makes it more likely that execution under the tool will observe the same behavior, including bugs, as native execution. Additionally, wrapping more naturally supports attaching to an application part-way through execution, where the application has already allocated heap objects.

B. Transparent Wrapping

Existing instrumentation platforms, including Pin [10] and Valgrind [14], provide support for wrapping functions. However, both Pin and Valgrind add an extra frame on the application stack, which violates *transparency*, i.e., it perturbs the semantics of the application and could cause an application to fail to execute correctly.

We identified and explored three different methods for wrapping functions transparently and robustly before settling on the third method for use in Dr. Memory. We require that the target function execute under control of the tool, with a *pre-hook* prior to the call and a *post-hook* after the call containing tool code that is executed natively. The pre-hook

is straightforward: there is only one entry point into a library function, and we simply watch for that address and invoke the pre-hook at that point. However, locating all return paths from the function for the post-hook is challenging.

The first technique we considered for locating return paths is to analyze the code when first encountered. By building a control-flow graph from the entry point, we can attempt to identify all the return points. We would then record each return instruction's address and invoke our post-hook whenever we reach one. However, it is not always easy or even possible to statically analyze an entire function. Hot/cold and other code layout optimizations, switches and other indirection, and mixed code and data all complicate analysis.

The second possible technique focuses on the call site. At a call instruction, we know the return point. If we can identify the call target we can set up for a post-hook for that target at the instruction after the call. If that instruction has already been executed and thus instrumented, we flush it from the tool's underlying code cache. To identify the target, direct calls as well as indirect calls utilizing standard library import mechanisms (PLT on Linux, IAT on Windows) are straightforward to handle. However, a call that indirects through an arbitrary register or memory location may not have a statically analyzable target. Extra work is required to support a general pre-hook that identifies the target, and a general post-hook for an indirect call that targets multiple wrapped routines.

The third technique, which we chose for Dr. Memory, takes a more dynamic and reactive approach to avoid the limitations of identifying either return instructions or call targets ahead of time. Once inside a target function itself, at the entry point, the return address is obtained and recorded for post-hook instrumentation when it is reached later. If the instruction at that address has already been executed, we flush it from the code cache. We can support multiple wrapped routines called from one site by storing which function is being returned from.

For all of these techniques, if wrapped routine A makes a tailcall to wrapped routine B, one of our post-hooks will be skipped (A for technique one, B for two and three). To solve the tailcall problem, we watch for a direct jump to a wrapped function and store the target. When we reach a post-hook, if a tailcall target is stored, we first perform the post-hook for that target before acting on the natural post-hook. A shadow stack could alternatively be used for handling tailcalls. A Windows exception unwind can also skip a post-hook by exiting a wrapped routine; we intercept exception handling to handle this case. A `longjmp` can be similarly handled.

C. Layered Heap Functions

Once our pre-hooks and post-hooks are in place, we can take appropriate actions prior to and after each heap library function, including modifying requested allocation sizes in order to add redzones and modifying return values to hide the redzones from the application. With layered heap calls, however, we need to be careful to only modify the arguments for the outer layer. We use a recursion counter to identify in which layer we are currently operating.

We must not report errors when heap library functions access heap headers or unallocated heap memory. We use a separate recursion counter for that purpose: a count that is incremented for all heap library routines, while our recursion count for argument adjustments is only incremented in the core library routines for which we need to modify arguments.

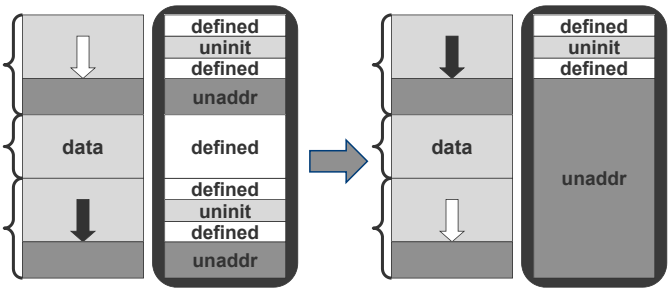

Fig. 3. The consequences of a stack swap being incorrectly interpreted as a stack de-allocation. The data in between these two stacks on the heap ends up marked as *unaddressable* after the swap from the lower stack to the upper stack is treated as a stack de-allocation within a single stack.

D. Statically Linked Libraries

On Windows, many applications are linked with a static copy of the C library. Identifying `malloc` library routines for these applications requires debugging information. The `PDB` debugging information format on Windows is not well documented and is best parsed using Microsoft's `dbghelp.dll`. That library in turn imports from five or more Windows API libraries. Using all of these libraries in the same process as the application requires a private loader that can isolate a second copy of a library from the application's copy. DynamoRIO provides this private loader for Dr. Memory's use. An alternative is to launch a separate process to read the symbols. While that can incur performance overhead from communication latency, a separate process can be necessary for applications that have extremely large debugging data that will not fit into the same process as the application.

E. Distinguishing Stack From Heap

Memory checking tools must monitor all stack pointer changes to mark the memory beyond the top of the stack as invalid and to mark newly allocated stack memory as uninitialized. However, unlike heap allocations, stack allocations do not normally use library routines (one exception is `alloca`). Instead, the stack pointer register is directly modified, either with a relative addition or subtraction, or by copying an absolute value from a frame pointer or elsewhere. When an absolute value is copied, it can be difficult to distinguish a stack pointer change on the same stack from a swap to a different stack. Stack swaps are used for lightweight threads (such as *fibers* on Windows) and other application constructs.

There are serious consequences to incorrectly interpreting a stack register change. Figure 3 illustrates one such scenario. The figure shows two stacks allocated on the heap and separated by non-stack data, along with the corresponding shadow memory for each of the three heap allocations. The arrows indicate the top of each stack. Initially the stack pointer register points at the top of the bottom stack, the filled-in arrow. If the application sets the stack pointer register to the top arrow,

resulting in the right-hand picture, and Dr. Memory interprets that as a de-allocation on one large stack, then we will mark all memory beyond the new top of stack as unaddressable. This will result in *false positive* errors reported on any subsequent access to the data in the middle or to legitimate stack entries on the bottom stack. Even worse, *false negatives* are possible when an error is made in the other direction: if a stack de-allocation is treated as a swap, accesses beyond the new top of stack may go undetected.

The example scenario in Figure 3 is a real situation that we have encountered in a version of the ISIS distributed resource manager [5]. Allocating stacks on the heap is not uncommon; the Cilk programming language is another example.

The approach taken by existing tools, including Memcheck, to distinguish swaps is to use a user-controllable *stack swap threshold* [16]. Any stack pointer change smaller than the threshold is treated as an allocation on the same stack, while any change above the threshold is considered a swap. Unfortunately, there is no single threshold that works for all applications, and it is up to the user to set it appropriately for a given application. However, deciding on the proper value is not straightforward. Furthermore, a single value may not work for all stacks within the same application.

We have developed a scheme for automatically adjusting the threshold that removes any need for manually specifying a value. We adjust the threshold in both directions, but as it is much easier to detect a too-small threshold than a too-large one, we start with a relatively small value of 36KB. For comparison, this is about fifty times smaller than Memcheck's default value.

Our instrumentation that handles stack register changes checks for absolute changes that are larger than the current threshold. For such values, we exit our fastpath instrumentation to a C routine that examines the memory allocation containing the top of the stack and verifies whether it is indeed a stack swap or not. We assume that no stack swap will use a relative change, and thus avoid checking for the threshold on many stack pointer adjustments. If the change is not a swap, we increment a counter. If the counter exceeds a small value (16 in our implementation), we increase the threshold, but linearly, by one page each time the counter is exceeded. This ameliorates performance impact from frequent large stack allocations, while keeping the threshold small, which is safer than having it too large. We measured our performance on the *gcc* SPEC2000 [18] benchmark, which contains a number of large stack allocations in the tens, hundreds, and even thousands of kilobytes, and our linear increase worked well.

Dr. Memory does not know which heap allocations or memory mappings are meant to be stacks, beyond the initial stack for each thread. Unknown stacks are identified when a stack allocation touches valid memory: for a known stack, memory beyond the current top of the stack is always *unaddressable*. When an unknown stack is found, we do not mark the rest of the memory region as invalid, because some applications use a single heap allocation to store not only a stack but also data adjacent to the stack. We instead mark the next lower page as invalid, and repeat one page at a time for each trigger. This heuristic avoids marking non-stack data as invalid unless the application stack gets within a page of its allocated bounds.

On detecting an unknown stack, if its size is smaller than our stack swap threshold, we reduce the threshold to the stack size minus one page. The goal here is to detect a stack swap between adjacent stacks. Typically, multiple nearby stacks are the same size. We can still incorrectly interpret a stack change as a stack swap if there are two adjacent stacks and the top of each is very near the border of the other stack. This is unlikely, however, and we have never observed it in practice. Without either reducing the swap threshold such that it impacts performance, or having semantic information from the application, it is not clear how to avoid this corner case.

V. Kernel Interaction

Most memory-checking tools operate on user-mode applications. While each of the application's memory references can be directly monitored, any memory in the application's address space that is read or written by the underlying operating system kernel is invisible to the tool. Thus, each system call's effects on application memory must be emulated in shadow memory. Failing to do so for memory read by system calls leads to false negatives: uninitialized values passed to the kernel will not be detected. Failing to emulate for memory written by system calls leads to false positives, where the tool will complain about uninitialized memory read by the application when in fact that memory was initialized by the kernel. Proper system call shadow emulation is essential to creating a usable memory checking tool.

A. Unknown System Calls

In order to emulate each system call's effects in shadow memory, the type and size of each system call parameter must be known. For Linux this information is readily available, but this is not an easy task on proprietary systems like Windows.

Windows contains two main sets of system calls: the `ntoskernl` system calls and the `win32k` system calls. The former are analogous to UNIX system calls. The latter are graphical display-related calls, as the graphics subsystem lives in the kernel for all recent Windows versions.

The `ntoskernl` system call parameters have been documented, if not officially, and Dr. Memory does explicit checking on each parameter. However, very little information is available about the `win32k` system calls. For these, Dr. Memory does not have a list of parameters for each call, and is forced to use techniques to identify the parameters dynamically. We focus on avoiding false positives by identifying output parameters.

Recall that Dr. Memory shadows both memory and registers with metadata indicating whether the application data is allocated and initialized. We use Dr. Memory's shadow metadata to identify potential system call parameters. Windows `ntoskernl` system calls have up to 17 parameters. For `win32k` system calls, we check the first 24 system call parameters and immediately ignore any that are not marked *defined*. For those that are, we then look for pointers to memory. If a parameter's value is an address that is not marked

as *unaddressable*, we start scanning the shadow values from that point forward. Because some parameters have both input and output components, we do not stop when we hit initialized data: we continue until we reach *unaddressable* memory or we hit a maximum size, which we have set at 2KB. We then make a copy of the original application memory. We also mark every *uninitialized* byte with a sentinel value so that we can detect kernel writes to data that happens to have the same value prior to the system call. This is most common with zero values.

User-level architectural simulators also need to detect system call changes so they can faithfully replay execution traces. The typical method is to use a separate process that keeps an identical copy of all of user-space memory [12]. After a system call, any changes can be identified. However, memory that happened to be the same value prior to the system call can be ignored for the purposes of replay, while we must update our shadow metadata for such memory. Our sentinel write allows us to detect all changes.

After the system call, we iterate over the *uninitialized* bytes. If a byte's value no longer matches the sentinel, we mark it as *defined*. If the byte does match the sentinel, we restore it to its pre-system-call value. There is a risk of the kernel writing the sentinel value, and a risk of races when any of the data under consideration is thread-shared. However, if another thread writes to the data, it will become defined anyway. Another concern is if this is heap memory and another thread frees the memory, but that is only likely when we misinterpreted a value that is not really a system call parameter: but since most of these parameters are on the stack and are thus thread-local variables, this should only happen with large integers that look like pointers. These are all risks that we tolerate.

In Dr. Memory, the scheme we have described removes 95% of false uninitialized read reports in graphical applications such as `calc.exe` on Windows. To eliminate the rest we plan to extend our scheme to locate secondary pointers. Additionally, there are system call changes that cannot be detected purely by examining system call parameters as memory pointers. Some system calls write data based on a handle or other indirect method of referring to memory. Without semantic information on these parameters, our only solution is to compare all of memory, which would be prohibitively expensive for our purposes. Today we rely on explicit handling for such cases.

VI. Leak Detection

Memory leaks are a common problem with C and C++ applications, and can be difficult to identify and track down without the help of a dedicated tool. Dr. Memory identifies *reachability-based leaks*, where a leak is defined as heap memory that no longer has any pointer to it, as opposed to considering any unfreed memory as a leak.

A. Leak Scan

At program exit time, or any intermediate point requested by the user, Dr. Memory suspends all of the application threads and performs a leak scan. This is similar to a garbage collector's mark-and-sweep operation. Without semantic information, large integer values cannot be distinguished from pointers, and are conservatively interpreted as pointers.

For its root set our leak scan uses the registers of each thread as well as all non-heap addressable memory, which includes below the top of the stack for each thread and the data section of each library. Only pointers marked as *defined* by Dr. Memory are considered. For efficiency, we also only consider aligned pointers. We have yet to observe any loss of precision from this decision.

Dr. Memory uses a feature called a *nudge* to allow the user to request a leak scan at any point. Nudges are sent from a separate process and notify the target application that is running under Dr. Memory that an action is requested. On Windows we implement nudges via injected threads; on Linux we use signals. In each case we set fields such that the nudge can be distinguished from a message meant for the application. Nudges can be used to help determine roughly when the last pointer to an allocation was lost, if the callstack of the allocation is not sufficient to pinpoint the error in the source code. Each nudge performs a full leak scan, and by nudging periodically the first occurrence of the leak can be identified.

B. Types of Leaks

Dr. Memory uses the general leak classification introduced by Purify [7] and divides all heap memory that is still allocated at the time it does its leak scan into three categories:

1) Memory that is still reachable by the application. This is *not* considered a leak. Many applications do not explicitly free memory whose lifetime matches the process lifetime and this is not considered an error by Dr. Memory.
2) Memory that is definitely not reachable by the application (at least, not by an aligned pointer to the start or middle of the allocated block). This is called a *leak* by Dr. Memory, as there is no way for the application to free this memory: it has lost all references to it.
3) Memory that is reachable only via pointers to the middle of the allocation, rather than the head. This is called a *possible leak* by Dr. Memory. These may or may not be legitimate pointers to that allocation. Several cases of known legitimate mid-allocation pointers are discussed in Section VI-C.

Each of the leak and possible leak categories is further broken down into direct and indirect leaks. An indirect leak is a heap object that is reachable by a pointer to its start address, but with all such pointers originating in leaked objects. Dr. Memory reports the number of leaks, possible leaks, and still-reachable allocations, along with the allocation callstack for each.

C. Possible Leak False Positives

The possible leak category is not always useful, as C++ applications can have many false positives in this category, leading users to ignore the category altogether. There are several C++ object layouts that result in legitimate pointers to the middle of an allocated heap object:

- For C++ arrays allocated via `new[]` whose elements have destructors, the `new[]` operator adds a header but returns to the caller the address past the header. Thus, at leak scan time, only a pointer to the middle of the heap object exists.
- For some C++ compilers, a pointer to an instance of a class with multiple inheritance that is cast to one of the parents can end up pointing to the sub-object parent representation in the middle of the allocation. We have observed large C++ applications with tens of thousands of possible leaks due to multiple inheritance.
- The `std::string` class places a `char[]` array in the middle of an allocation and stores a pointer to the array.

Unlike any existing tool we are aware of, Dr. Memory recognizes each of these classes of common mid-allocation pointers and eliminates them from the list of possible leaks, increasing the signal-to-noise ratio of the possible leak category. Our identification methods do not rely on having symbol information, but if such information is available we could use it to augment our checks.

To identify `new[]`, if we see a mid-allocation pointer during our leak scan that points one `size_t` inside the block where the value stored at the head of the block equals the block size, we consider the block fully reachable and do not place it in the possible leak category. This more-general check will also support an application that wraps each `malloc` block with its own header that stores just the size, which we have observed in practice.

For multiple inheritance, we look for a vtable pointer at both the pointed-at mid-allocation point and at the block head. We assume here that the C++ implementation stores the vtable pointer as the hidden first field, which is the case for all compilers we have encountered. We safely de-reference both potential vtable pointers. We first check whether these are clearly small integers rather than pointers. This check improves performance significantly, ruling out nearly all non-vtable instances. If both of the two vtable pointers pass, we then check whether each location pointed at contains a table of functions. We ignore alignment, as Windows vtables often point into the ILT and are not aligned; primarily we check whether each entry points into the text section of a library or the executable. The function pointers do not need to all point to the same library. We simply march through the potential table, skipping zero entries, until we find at least two function pointers. At that point we conclude that this is in fact a vtable. This scheme works well in practice and we have yet to find a false positive.

An instance of the `std::string` class is identified by looking for its specific layout: the mid-allocation pointer points to a character array that follows three header fields, length, capacity, and a reference count, at the head of the block.

D. Storing Callstacks Efficiently

Each memory leak report includes its allocation callstack. This necessitates recording the callstack for every live allocation, since any allocation could later be leaked. Some applications have millions of simultaneously live allocations. If callstacks are stored as text strings, they can average several hundred bytes per callstack, resulting in hundreds of megabytes or more of memory used just for callstacks.

To reduce the memory required, we use a binary callstack encoding. Additionally, because many allocations share the same callstack, we store callstacks in a hashtable to avoid storing duplicate copies. Each callstack consists of a variable number of frames which we dynamically allocate.

Each frame contains an absolute address and a module and offset (we do not store the frame pointer stack addresses). Since the same module could be loaded at two different locations during the course of execution, we must store both the name, which is a pointer into a module name hashtable from which entries are never removed, and the offset. We pack further using an array of names so we can store an index in the frame that is a single byte, optimizing for the common case of less than 256 libraries. This index shares a 32-bit field with the module offset, which is then limited to 16MB. For modules larger than 16MB, we use multiple entries that are adjacent in the module name array. The hashtable holds the index of the first such entry, allowing us to reconstruct the actual offset. Each frame thus takes up only 8 bytes for 32-bit applications.

Our compacted binary callstack encoding that shares callstacks via a hashtable uses a small fraction of the memory required for non-shared callstacks in text format, reducing usage on large applications from several hundred megabytes to a dozen or so, a huge savings (see Table II).

VII. Experimental Results

We evaluated Dr. Memory on the SPEC CPU2006 benchmark suite [19] with reference input sets. We omit `481.wrf` as it fails to run natively. The benchmarks were compiled as 32-bit using gcc 4.3.2 -O2. We ran our Linux experiments on a six-core, dual-processor Intel Core i7 with 2.67GHz clock rate, 12MB L2 cache, and 48GB RAM, running Debian 5.0.7. For Windows we compiled the C and C++ benchmarks with Microsoft Visual Studio 2005 SP1 and ran our experiments on an Intel Core 2 Quad Q9300 with 2.50GHz clock rate, 6MB L2 cache, and 4GB RAM, running Windows Vista SP1.

To compare to Memcheck, we used the latest Valgrind release, version 3.6.0. Memcheck is unable to run `434.zeusmp` and `447.dealII`. The 3.6.0 release is also unable to run `410.bwaves`, though by increasing `VG_N_SEGMENTS` in the Memcheck source code from 5000 to 6000 and re-compiling it succeeds. On `400.perlbench`, Memcheck runs out of memory toward the end of one of the three runs; we estimate the total runtime as the runtime to that point.

A. Comparative Performance

We evaluated Dr. Memory on Linux where a direct comparison with Valgrind's Memcheck is possible. Figure 4 shows that Dr. Memory is twice as fast as Memcheck on average, and up to four times faster on individual benchmarks. A faster tool is a more usable and practical tool. Dr. Memory's superior performance stems from the use of fastpath kernels and the collection of optimizations described in Section III.

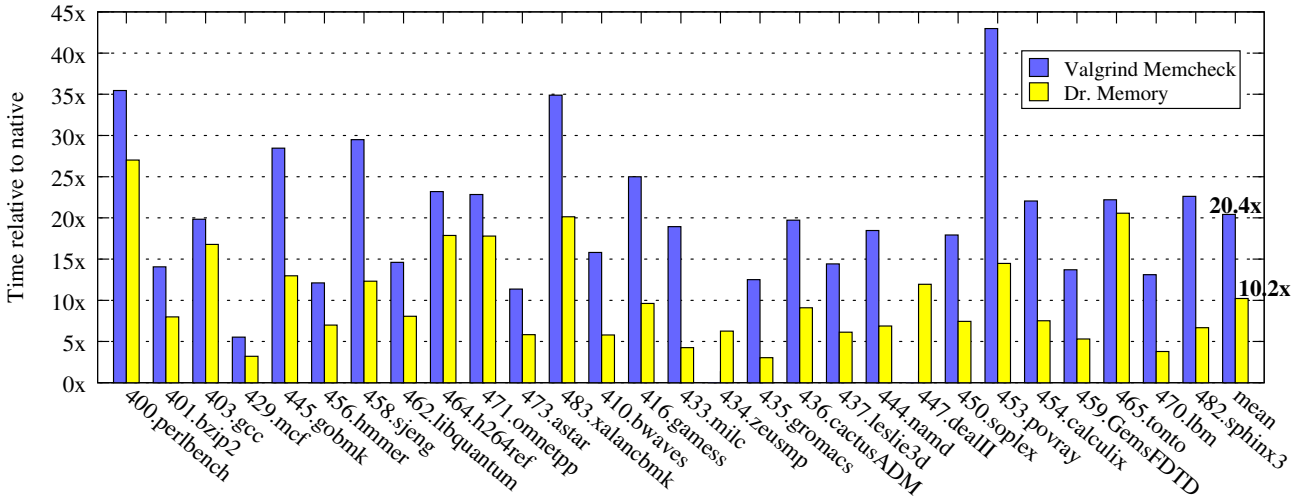

Fig. 4. The performance of Dr. Memory compared to Valgrind Memcheck [16] on the SPEC CPU2006 benchmarks on 32-bit Linux. On average, Dr. Memory is twice as fast as Memcheck, and is up to four times faster on individual benchmarks. Memcheck is unable to run `434.zeusmp` and `447.dealII`.

The two tools' base systems, DynamoRIO and Valgrind, differ significantly in performance when executing with no tool. DynamoRIO is several times faster than Valgrind with no tool (*Nulgrind*). However, this relative performance has little bearing when comparing the memory checking tools, for two reasons. First, a negligible amount of time is spent in the base system when running a 10x–20x slowdown tool. A memory checking tool executes about 15–30 new instructions for each instruction executed by the base system. DynamoRIO does not make these new instructions faster. 95% or more of execution time is spent in these added instructions: i.e., little time is spent doing what the base system with no tool does.

Furthermore, the impact of the base system can be in the *opposite* direction from what a base comparison would imply: Valgrind's slowdown with no tool stems from its emulation of the application's state, versus DynamoRIO's mapping of the application state directly to the native machine state. When significant instrumentation is added, a DynamoRIO tool must perform additional context switches beyond what a Valgrind tool must do, because the base Valgrind system is already doing that work. This makes the base system comparison an invalid predictor of relative tool performance.

Like Memcheck, Dr. Memory implements per-byte shadowing and uses shadow propagation to avoid false positives, unlike many tools such as Purify [7] and Intel Parallel Inspector [9] that do not propagate shadow values and thus incur less overhead but give up accuracy as a result.

Unlike Dr. Memory, Memcheck supports per-bit shadowing, but only switches from per-byte on an as-needed basis, which is typically for less than 0.1% of memory accesses [13]. Few benchmarks in our suite actually contain bitfields. Thus, the presence of per-bit shadowing in Memcheck should have little effect on its performance here.

B. Windows Performance

Figure 5 shows Dr. Memory's performance on Windows. Overall the performance is comparable to our Linux performance. However, the different compiler and different libraries

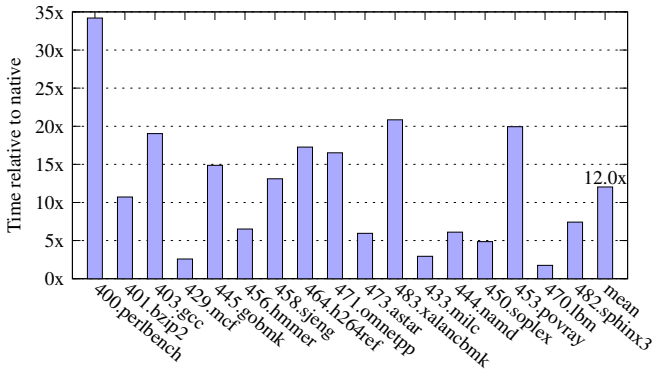

Fig. 5. The performance of Dr. Memory relative to native on Windows on the SPEC CPU2006 benchmarks that can be compiled by the Microsoft Visual Studio compiler.

can result in different performance bottlenecks and results for the same benchmark on the two platforms. For example, `alloca` is more frequently employed on Windows, requiring a fast method of allowing its probes beyond the top of the stack, while on Linux `alloca`'s performance does not matter. The instruction mix used by the compilers differs as well. For example, the `cmpxchg8b` instruction is seen much more frequently on Windows; if not handled in the fastpath, it can be a performance bottleneck.

C. Component Performance

Figure 6 shows the performance impact of each of the optimizations presented in Section III-C. Using a table lookup for addressability has a sizeable impact as it keeps partially-undefined word references on the fastpath. Using faults for special blocks keeps the instrumentation size small, and whole-bb spilling reduces the number of register spills and restores significantly. The other optimizations have less dramatic but still significant impacts.

We have found that the sources of performance bottlenecks for Dr. Memory can vary widely across different applications. If enough instructions of a certain type fall into Dr. Memory's

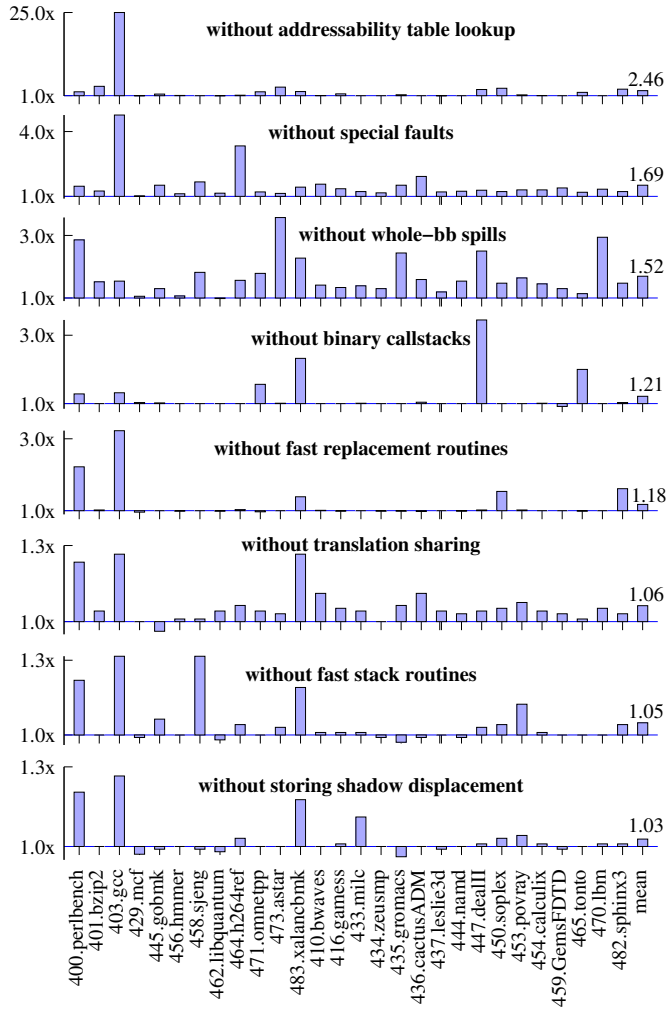

Fig. 6. Performance impact of the eight Dr. Memory optimizations described in Section III-C on the SPEC CPU2006 benchmarks. Each graph shows the slowdown versus the fully optimized Dr. Memory that is incurred when that optimization is disabled.

slowpath, performance will suffer. For example, `465.tonto` contains many ten-byte floating point operations. Initially these were not handled in our fastpath, and tonto's performance was poor. We had not previously seen this behavior. For `400.perlbench`, processing each `malloc` and `free` tends to dominate the runtime. Other examples on Windows were given in the previous section.

D. Memory Usage

Table II shows the memory used for storing callstacks when using text, binary, and our compacted binary format with and without sharing. Both sharing and a binary format are required to avoid excessive memory consumption, and our compacted binary encoding described in Section VI-D reduces memory usage further. Additionally, producing a text format takes additional time, as previously seen in Figure 6. The most dramatic impact is on `447.dealII`, which has 1.3 million simultaneously active allocations, most of which were allocated from identical call sites, producing ample opportunities for sharing callstacks.

	Non-shared		Shared	
Benchmark	**Text**	**Binary**	**Binary**	**Compacted**
400.perlbench	1,871,303	442,996	474	326
401.bzip2	5	2	2	2
403.gcc	108,726	26,365	962	658
429.mcf	4	2	2	2
445.gobmk	708	177	10	5
456.hmmer	455	122	8	6
458.sjeng	4	2	2	2
462.libquantum	4	2	2	2
464.h264ref	5,592	1,408	21	15
471.omnetpp	1,415,199	331,926	910	637
473.astar	130,435	35,777	4	3
483.xalancbmk	1,205,862	290,176	329	228
410.bwaves	16	5	5	4
416.gamess	21	7	4	3
433.milc	20	6	3	3
434.zeusmp	34	10	10	7
435.gromacs	1,162	282	29	21
436.cactusADM	2,956	707	583	408
437.leslie3d	16	5	5	4
444.namd	407	120	5	4
447.dealII	*3,154,396+*	*785,832+*	53	38
450.soplex	67	18	18	13
453.povray	7,499	1,762	77	54
454.calculix	14,387	3,430	28	20
459.GemsFDTD	71	19	19	14
465.tonto	9,410	2,222	26	19
470.lbm	4	2	2	2
482.sphinx3	0	0	0	0

Table II. Memory usage, in KB, of callstacks used to store allocation sites and to store reported error locations. For benchmarks with multiple runs, the largest run is shown. The first two columns show the enormous amounts of memory needed when non-shared text or binary callstacks are used. The third column shows the usage of shared binary callstacks, and the final column gives the results of our approach of shared and compacted binary callstacks. With non-shared callstacks, `447.dealII` runs out of memory and fails to finish running.

E. Accuracy

Without our unknown system call handling in place, running the Windows `calc.exe` graphical calculator results in 105 unique and 1424 total uninitialized reads reported. Using the algorithm in Section V-A, nearly all of those false positives are eliminated and only 4 remain. We believe we can eliminate those as well with further work.

Without our possible leak heuristics, one large proprietary C++ application contained 24,235 possible leak allocation sites, each with several leaked blocks, a daunting number for a user to analyze. After eliminating the categories of legitimate possible leaks described in Section VI-C, Dr. Memory reported only 6 possible leak sites in the application. For the benchmark `403.gcc`, Dr. Memory reports an average of 20 possible sites for each of the benchmark's 9 runs, while Memcheck averages 596 possible sites.

Dr. Memory has identified several unaddressable errors in a version of the Small Footprint CIM Broker (SFCB) [1], including a race condition on a library unload, a use-after-free error, and an out-of-bounds access, along with numerous leaks in SFCB and in a version of the ISIS distributed resource

manager [5]. We have also found three uninitialized reads in the main Cygwin library, an uninitialized read in a Windows system library, and an uninitialized read and an out-of-bounds memory access in Linux glibc.

VIII. Related Work

Perhaps the most widely-used memory checking tool today is MemCheck [16], built on the Valgrind [14] dynamic instrumentation platform. Dr. Memory's shadow states, shadow combination rules, and propagation are all similar to those of Memcheck [13]. Memcheck only supports UNIX platforms. It replaces library allocation functions, uses a single threshold for stack swap detection, and does not distinguish false positive possible leaks from common C++ data layouts, making its possible leak reports more difficult to use.

Purify [7] was one of the first commercial memory checking tools, and the first tool to combine detection of memory leaks with detection of use-after-free errors. Purify uses link-time instrumentation and reports reads of uninitialized errors immediately, which can result in false positives. Purify's basic leak detection approach is used by both Memcheck and Dr. Memory.

Parallel Inspector [9] is a commercial tool built on the Pin [10] dynamic instrumentation platform that combines data race detection with memory checking. Like Purify, it reports reads of uninitialized errors immediately. Details of its implementation are not publicly available.

Insure++ [15] is another commercial memory checking tool. It supports inserting instrumentation at various points, including the source code prior to compile time, at link time, and at runtime, but its more advanced features require source code instrumentation. Instrumentation at runtime is inserted using dynamic binary instrumentation, just like Dr. Memory, Memcheck, and Parallel Inspector, via a tool called Chaperon. Insure++ does support delaying reports of uninitialized memory but only across copies and not other operations.

Third Degree [8] is a memory checking tool for the Alpha platform. It inserts instrumentation at link time using ATOM [17]. It detects uninitialized reads by filling newly allocated memory with a sentinel or canary value and reporting an error on any read of the canary value.

BoundsChecker [11] monitors Windows heap library calls and detects memory leaks and unaddressable accesses. It does not detect uninitialized reads.

Some leak detection tools, including LeakTracer [2] and mprof [6], only report memory that has not been freed at the end of execution. For these tools to be usable, the application must free all of its memory prior to exiting, even though it may have data whose lifetime is the process lifetime where it is more efficient to let the operating system free those resources. Reachability-based leak detection, in contrast, uses a memory scan that is similar to a mark-and-sweep garbage collector [3] to identify orphaned memory allocations that can no longer be accessed. This type of leak detection is used by most modern memory checking tools, including ours.

IX. Conclusion

Memory checking tools are invaluable for detecting memory errors in applications. However, such tools are difficult to build due to three significant challenges: performance, accuracy, and system dependencies. This paper presents a memory checking tool for both Windows and Linux that addresses these challenges. It handles undocumented Windows system calls and the complex Windows heap API, and avoids reporting false positive memory leaks stemming from common C++ data layouts that fool traditional leak locating algorithms. Dr. Memory employs efficient instrumentation techniques and optimizations and out-performs the state-of-the-art Valgrind Memcheck tool by an average of 2x. These improvements combine to create a more practical tool. Dr. Memory is open-source and available at `http://code.google.com/p/drmemory/`.

References

[1] "Small footprint CIM broker (SFCB)," `http://sourceforge.net/apps/mediawiki/sblim/index.php?title=Sfcb`.

[2] E. Andreasen and H. Zeller, "Leaktracer," Aug. 2003, `http://www.andreasen.org/LeakTracer/`.

[3] H. Boehm, A. Demers, and M. Weiser, "A garbage collector for C and C++," `http://www.hpl.hp.com/personal/Hans_Boehm/gc/`.

[4] D. Bruening, "Efficient, transparent, and comprehensive runtime code manipulation," Ph.D. dissertation, M.I.T., Sep. 2004.

[5] T. Clark and K. Birman, "Using the ISIS resource manager for distributed, fault-tolerant computing," Tech. Rep. TR 92-1289, Jun. 1992.

[6] B. Z. Department, B. Zorn, and P. Hilfinger, "A memory allocation profiler for C and Lisp programs," in *Proc. of the Summer 1988 USENIX Conference*, 1988, pp. 223–237.

[7] R. Hastings and B. Joyce, "Purify: Fast detection of memory leaks and access errors," in *Proc. of the Winter USENIX Conference*, Jan. 1992, pp. 125–136.

[8] Hewlett-Packard, "Third Degree," `http://h30097.www3.hp.com/developerstoolkit/tools.html`.

[9] Intel, "Intel Parallel Inspector," `http://software.intel.com/en-us/intel-parallel-inspector/`.

[10] 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 *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*, Jun. 2005, pp. 190–200.

[11] Micro Focus, "Boundschecker," `http://www.microfocus.com/products/micro-focus-developer/devpartner/visual-c.aspx`.

[12] S. Narayanasamy, C. Pereira, H. Patil, R. Cohn, and B. Calder, "Automatic logging of operating system effects to guide application-level architecture simulation," in *Proc. of the Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '06/Performance '06)*, 2006, pp. 216–227.

[13] N. Nethercote and J. Seward, "How to shadow every byte of memory used by a program," in *Proc. of the 3rd International Conference on Virtual Execution Environments (VEE '07)*, Jun. 2007, pp. 65–74.

[14] ——, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*, Jun. 2007, pp. 89–100.

[15] Parasoft, "Insure++," `http://www.parasoft.com/jsp/products/insure.jsp?itemId=63`.

[16] J. Seward and N. Nethercote, "Using Valgrind to detect undefined value errors with bit-precision," in *Proc. of the USENIX Annual Technical Conference*, 2005, pp. 2–2.

[17] A. Srivastava and A. Eustace, "ATOM: A system for building customized program analysis tools," in *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '94)*, Jun. 1994, pp. 196–205.

[18] Standard Performance Evaluation Corporation, "SPEC CPU2000 benchmark suite," 2000, `http://www.spec.org/osg/cpu2000/`.

[19] ——, "SPEC CPU2006 benchmark suite," 2006, `http://www.spec.org/osg/cpu2006/`.