AddressSanitizer: A Fast Address Sanity Checker
(This paper has been accepted to USENIX ATC 2012)

Konstantin Serebryany, Derek Bruening, Alexander Potapenko, Dmitry Vyukov
Google
{*kcc,bruening,glider,dvyukov*}*@google.com*

Abstract

Memory access bugs, including buffer overflows and uses of freed heap memory, remain a serious problem for programming languages like C and C++. Many memory error detectors exist, but most of them are either slow or detect a limited set of bugs, or both.

This paper presents AddressSanitizer, a new memory error detector. Our tool finds out-of-bounds accesses to heap, stack, and global objects, as well as use-after-free bugs. It employs a specialized memory allocator and code instrumentation that is simple enough to be implemented in any compiler, binary translation system, or even in hardware.

AddressSanitizer achieves efficiency without sacrificing comprehensiveness. Its average slowdown is just 73% yet it accurately detects bugs at the point of occurrence. It has found over 300 previously unknown bugs in the Chromium browser and many bugs in other software.

1 Introduction

Dozens of memory error detection tools are available on the market [3, 8, 11, 13, 15, 21, 23, 24]. These tools differ in speed, memory consumption, types of detectable bugs, probability of detecting a bug, supported platforms, and other characteristics. Many tools succeed in detecting a wide range of bugs but incur high overhead, or incur low overhead but detect fewer bugs. We present AddressSanitizer, a new tool that combines performance and coverage. AddressSanitizer finds out-of-bounds accesses (for heap, stack, and global objects) and uses of freed heap memory at the relatively low cost of 73% slowdown and 3.4x increased memory usage, making it a good choice for testing a wide range of C/C++ applications.

AddressSanitizer consists of two parts: an instrumentation module and a run-time library. The instrumentation module modifies the code to check the *shadow state* for each memory access and creates *poisoned redzones* around stack and global objects to detect overflows and underflows. The current implementation is based on the LLVM [4] compiler infrastructure. The run-time library replaces `malloc`, `free` and related functions, creates poisoned redzones around allocated heap regions, delays the reuse of freed heap regions, and does error reporting.

1.1 Contributions

In this paper we:

- show that a memory error detector can leverage the comprehensiveness of shadow memory with much lower overhead than the conventional wisdom;
- present a novel shadow state encoding that enables compact shadow memory – as much as a 128-to-1 mapping – for detecting out-of-bounds and use-after-free bugs;
- describe a specialized memory allocator targeting our shadow encoding;
- evaluate a new publicly available tool that efficiently identifies memory bugs.

1.2 Outline

After summarizing related work in the next section, we describe the AddressSanitizer algorithm in Section 3. Experimental results with AddressSanitizer are provided in Section 4. We discuss further improvements in Section 5 and then conclude the paper.

2 Related Work

This section explores the range of existing memory detection tools and techniques.

2.1 Shadow Memory

Many different tools use *shadow memory* to store metadata corresponding to each piece of application data. Typically an application address is mapped to a shadow address either by a direct scale and offset, where the full application address space is mapped to a single shadow address space, or by extra levels of translation involving table lookups. Examples of direct mapping include TaintTrace [10] and LIFT [26]. TaintTrace requires a shadow space of equal size to the application address space, which results in difficulties supporting applications that cannot survive with only one-half of their normal address space. The shadow space in LIFT is one-eighth of the application space.

To provide more flexibility in address space layout, some tools use multi-level translation schemes. Valgrind [20] and Dr. Memory [8] split their shadow memory into pieces and use a table lookup to obtain the shadow address, requiring an extra memory load. For 64-bit platforms, Valgrind uses an additional table layer for application addresses not in the lower 32GB.

Umbra [30, 31] combines layout flexibility with efficiency, avoiding a table lookup via a non-uniform and dynamically-tuned scale and offset scheme. BoundLess [9] stores some of its metadata in 16 higher bits of 64-bit pointers, but falls back to more traditional shadow memory on the slow path. LBC [12] performs a fast-path check using special values stored in the application memory and relies on two-level shadow memory on the slow path.

2.2 Instrumentation

A large number of memory error detectors are based on binary instrumentation. Among the most popular are Valgrind (Memcheck) [21], Dr. Memory [8], Purify[13], BoundsChecker [17], Intel Parallel Inspector [15] and Discover [23]. These tools find out-of-bounds and use-after-free bugs for heap memory with (typically) no false positives. To the best of our knowledge none of the tools based on binary instrumentation can find out-of-bounds bugs in the stack (other than beyond the top of the stack) or globals. These tools additionally find uninitialized reads.

Mudflap [11] uses compile-time instrumentation and hence is capable of detecting out-of-bounds accesses for stack objects. However, it does not insert redzones between different stack objects in one stack frame and will thus not detect all stack buffer overflow bugs. It is also known to have false positive reports in complex C++ code[1].

[1] `http://gcc.gnu.org/bugzilla/show_bug.cgi?id=19319`

CCured [19] combines instrumentation with static analysis (only for C programs) to eliminate redundant checks; their instrumentation is incompatible with uninstrumented libraries.

LBC [12] uses source-to-source transformation and relies on CCured to eliminate redundant checks. LBC is limited to the C language and does not handle use-after-free bugs.

Insure++ [24] relies mainly on compile-time instrumentation but also uses binary instrumentation. Details of its implementation are not publicly available.

2.3 Debug Allocators

Another class of memory error detectors uses a specialized memory allocator and does not change the rest of the execution.

Tools like Electric Fence [25], Duma [3], GuardMalloc [16] and Page Heap [18] use CPU page protection. Each allocated region is placed into a dedicated page (or a set of pages). One extra page at the right (and/or at the left) is allocated and marked as inaccessible. A subsequent page fault accessing these pages is then reported as an out-of-bounds error. These tools incur large memory overheads and may be very slow on `malloc`-intensive applications (as each `malloc` call requires at least one system call). Also, these tools may miss some classes of bugs (e.g., reading the byte at offset 6 from the start of a 5-byte memory region). If a bug is reported, the responsible instruction is provided in the error message.

Some other `malloc` implementations, including DieHarder [22] (a descendant of DieHard [5] `malloc`) and Dmalloc [2], find memory bugs on a probabilistic and/or delayed basis. Their modified `malloc` function adds redzones around memory regions returned to the user and populates the newly allocated memory with special *magic* values. The `free` function also writes magic values to the memory region.

If a magic value is read then the program has accessed an out-of-bounds or uninitialized value. However, there is no immediate detection of this. Through properly selected magic values, there is a chance that the program will behave incorrectly in a way detectable by existing application tests (DieHard [5] has a replicated mode in which it is able to detect such incorrect behavior by comparing the output of several program replicas initialized with different magic values). In other words, the detection of out-of-bounds reads and read-after-free bugs is probabilistic.

If a magic value in a redzone is overwritten, this will later be detected when the redzone is examined on `free`, but the tool does not know exactly when the out-of-bounds write or write-after-free occurred. For large programs it is often equivalent to reporting “your program

has a bug". Note that the goal of DieHarder is not only to detect bugs, but also to protect from security attacks.

The two debug `malloc` approaches are often combined. Debug `malloc` tools do not handle stack variables or globals.

The same magic value technique is often used for buffer overflow protection. StackGuard [29] and ProPolice [14] (the StackGuard reimplementation currently used by GCC) place a canary value between the local variables and the return address in the current stack frame and check for that value's consistency upon function exit. This helps to prevent stack smashing buffer overflows, but is unable to detect arbitrary out-of-bounds accesses to stack objects.

3 AddressSanitizer Algorithm

From a high level, our approach to memory error detection is similar to that of the Valgrind-based tool AddrCheck [27]: use shadow memory to record whether each byte of application memory is safe to access, and use instrumentation to check the shadow memory on each application load or store. However, AddressSanitizer uses a more efficient shadow mapping, a more compact shadow encoding, detects errors in stack and global variables in addition to the heap and is an order of magnitude faster than AddrCheck. The following sections describe how AddressSanitizer encodes and maps its shadow memory, inserts its instrumentation, and how its run-time library operates.

3.1 Shadow Memory

The memory addresses returned by the `malloc` function are typically aligned to at least 8 bytes. This leads to the observation that any aligned 8-byte sequence of application heap memory is in one of 9 different states: the first k $(0 \leq k \leq 8)$ bytes are addressable and the remaining $8-k$ bytes are not. This state can be encoded into a single byte of shadow memory.

AddressSanitizer dedicates one-eighth of the virtual address space to its shadow memory and uses a direct mapping with a scale and offset to translate an application address to its corresponding shadow address. Given the application memory address `Addr`, the address of the shadow byte is computed as `(Addr>>3)+Offset`. If `Max-1` is the maximum valid address in the virtual address space, the value of `Offset` should be chosen in such a way that the region from `Offset` to `Offset+Max/8` is not occupied at startup. Unlike in Umbra [31], the `Offset` must be chosen statically for every platform, but we do not see this as a serious limitation. On a typical 32-bit Linux or MacOS system, where the virtual address space is `0x00000000-0xffffffff`, we use `Offset = 0x20000000` (2^{29}). On a 64-bit system with 47 significant address bits we use `Offset = 0x0000100000000000` (2^{44}). In some cases (e.g., with `-fPIE/-pie` compiler flags on Linux) a zero offset can be used to simplify instrumentation even further.

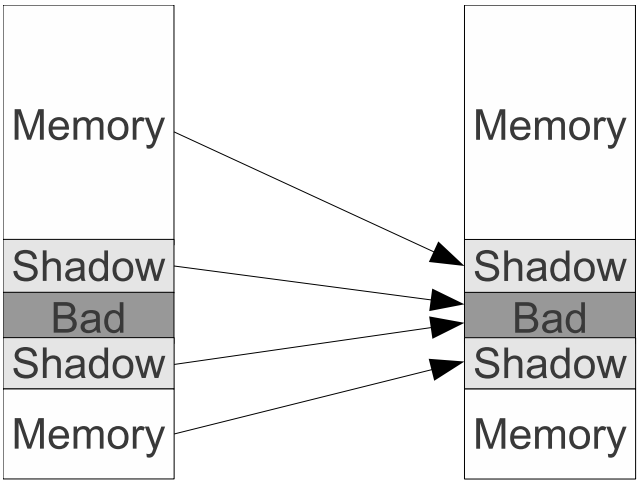

Figure 1: AddressSanitizer memory mapping.

Figure 1 shows the address space layout. The application memory is split into two parts (low and high) which map to the corresponding shadow regions. Applying the shadow mapping to addresses in the shadow region gives us addresses in the *Bad* region, which is marked inaccessible via page protection.

We use the following encoding for each shadow byte: 0 means that all 8 bytes of the corresponding application memory region are addressable; k $(1 \leq k \leq 7)$ means that the first k bytes are addressible; any negative value indicates that the entire 8-byte word is unaddressable. We use different negative values to distinguish between different kinds of unaddressable memory (heap redzones, stack redzones, global redzones, freed memory).

This shadow mapping could be generalized to the form `(Addr>>Scale)+Offset`, where `Scale` is one of $1 \ldots 7$. With `Scale=N`, the shadow memory occupies $1/2^N$ of the virtual address space and the minimum size of the redzone (and the `malloc` alignment) is 2^N bytes. Each shadow byte describes the state of 2^N bytes and encodes 2^N+1 different values. Larger values of `Scale` require less shadow memory but greater redzone sizes to satisfy alignment requirements. Values of `Scale` greater than 3 require more complex instrumentation for 8-byte accesses (see Section 3.2) but provide more flexibility with applications that may not be able to give up a single contiguous one-eighth of their address space.

3.2 Instrumentation

When instrumenting an 8-byte memory access, AddressSanitizer computes the address of the corresponding shadow byte, loads that byte, and checks whether it is zero:

```
ShadowAddr = (Addr >> 3) + Offset;
if (*ShadowAddr != 0)
  ReportAndCrash(Addr);
```

When instrumenting 1-, 2-, or 4- byte accesses, the instrumentation is slightly more complex: if the shadow value is positive (i.e., only the first k bytes in the 8-byte word are addressable) we need to compare the 3 last bits of the address with k.

```
ShadowAddr = (Addr >> 3) + Offset;
k = *ShadowAddr;
if (k != 0 && ((Addr & 7) + AccessSize > k))
    ReportAndCrash(Addr);
```

In both cases the instrumentation inserts only one memory read for each memory access in the original code. We assume that an N-byte access is aligned to N. AddressSanitizer may miss a bug caused by an unaligned access, as described in Section 3.5.

We placed the AddressSanitizer instrumentation pass at the very end of the LLVM optimization pipeline. This way we instrument only those memory accesses that survived all scalar and loop optimizations performed by the LLVM optimizer. For example, memory accesses to local stack objects that are optimized away by LLVM will not be instrumented. At the same time we don't have to instrument memory accesses generated by the LLVM code generator (e.g., register spills).

The error reporting code (`ReportAndCrash(Addr)`) is executed at most once, but is inserted in many places in the code, so it still must be compact. Currently we use a simple function call (see examples in Appendix A). Another option would be to use an instruction that generates a hardware exception.

3.3 Run-time Library

The main purpose of the run-time library is to manage the shadow memory. At application startup the entire shadow region is mapped so that no other part of the program can use it. The *Bad* segment of the shadow memory is protected. On Linux the shadow region is always unoccupied at startup so the memory mapping always succeeds. On MacOS we need to disable the address space layout randomization (ASLR). Our preliminary experiments show that the same shadow memory layout also works on Windows.

The `malloc` and `free` functions are replaced with a specialized implementation. The `malloc` function allocates extra memory, the redzone, around the returned region. The redzones are marked as unaddressable, or *poisoned*. The larger the redzone, the larger the overflows or underflows that will be detected.

The memory regions inside the allocator are organized as an array of freelists corresponding to a range of object sizes. When a freelist that corresponds to a requested object size is empty, a large group of memory regions with their redzones is allocated from the operating system (using, e.g., `mmap`). For n regions we allocate $n+1$ redzones, such that the right redzone of one region is typically a left redzone of another region:

rz1	mem1	rz2	mem2	rz3	mem3	rz4

The left redzone is used to store the internal data of the allocator (such as the allocation size, thread ID, etc.); consequently, the minimum size of the heap redzone is currently 32 bytes. This internal data can not be corrupted by a buffer underflow, because such underflows are detected immediately prior to the actual store (if the underflow happens in the instrumented code).

The `free` function poisons the entire memory region and puts it into *quarantine*, such that this region will not be allocated by `malloc` any time soon. Currently, the quarantine is implemented as a FIFO queue which holds a fixed amount of memory at any time.

By default, `malloc` and `free` record the current call stack in order to provide more informative bug reports. The `malloc` call stack is stored in the left redzone (the larger the redzone, the larger the number of frames that can be stored) while the `free` call stack is stored in the beginning of the memory region itself.

Section 4.3 discusses how to tune the run-time library.

3.4 Stack And Globals

In order to detect out-of-bounds accesses to globals and stack objects, AddressSanitizer must create poisoned redzones around such objects.

For globals, the redzones are created at compile time and the addresses of the redzones are passed to the run-time library at application startup. The run-time library function poisons the redzones and records the addresses for further error reporting.

For stack objects, the redzones are created and poisoned at run-time. Currently, redzones of 32 bytes (plus up to 31 bytes for alignment) are used. For example, given a program

```
void foo() {
  char a[10];
  <function body> }
```

the transformed code will look like

```
void foo() {
  char rz1[32]
  char arr[10];
  char rz2[32-10+32];
  unsigned *shadow =
    (unsigned*)(((long)rz1>>8)+Offset);
  // poison the redzones around arr.
  shadow[0] = 0xffffffff;  // rz1
  shadow[1] = 0xffff0200;  // arr and rz2
  shadow[2] = 0xffffffff;  // rz2
  <function body>
  // un-poison all.
  shadow[0] = shadow[1] = shadow[2] = 0;  }
```

3.5 False Negatives

The instrumentation scheme described above may miss a very rare type of bug: an unaligned access that is partially out-of-bounds. For example:

```
int *a = new int[2]; // 8-aligned
int *u = (int*)((char*)a + 6);
*u = 1;  // Access to range [6-9]
```

Currently we ignore this type of bug since all solutions we have come up with slow down the common path. Solutions we considered include:

- check at run-time whether the address is unaligned;
- use a byte-to-byte shadow mapping (feasible only on 64-bit systems);
- use a more compact mapping (e.g., `Scale=7` from Section 3.1) to minimize the probability of missing such a bug.

AddressSanitizer may also miss bugs in the following two cases (tools like Valgrind or Dr. Memory have the same problems). First, if an out-of-bounds access touches memory too far away from the object bound it may land in a different valid allocation and the bug will be missed.

```
char *a = new char[100];
char *b = new char[1000];
a[500] = 0;  // may end up somewhere in b
```

All out-of-bounds accesses within the heap redzone will be detected with 100% probability. If the memory footprint is not a serious constraint we recommend using large redzones of up to 128 bytes.

Second, a use-after-free may not be detected if a large amount of memory has been allocated and deallocated between the "free" and the following use.

```
char *a = new char[1 << 20];  // 1MB
delete [] a;  // <<< "free"
char *b = new char[1 << 28];  // 256MB
delete [] b;  // drains the quarantine queue.
char *c = new char[1 << 20];  // 1MB
a[0] = 0;     // "use". May land in 'c'.
```

3.6 False Positives

In short, AddressSanitizer has no false positives. However, during the AddressSanitizer development and deployment we have seen a number of undesirable error reports described below, all of which are now fixed.

3.6.1 Conflict With Load Widening

A very common compiler optimization called *load widening* conflicts with AddressSanitizer instrumentation. Consider the following C code:

```
struct X { char a, b, c; };
void foo() {
  X x; ...
  ... = x.a + x.c; }
```

In this code, the object `x` has size 3 and alignment 4 (at least). Load widening transforms `x.a+x.c` into one 4-byte load, which partially crosses the object boundary. Later in the optimization pipeline AddressSanitizer instruments this 4-byte load which leads to a false positive. To avoid this problem we partially disabled load widening in LLVM when AddressSanitizer instrumentation is enabled. We still allow widening `x.a+x.b` into a 2-byte load, because such a transformation will not cause false positives and will speedup the instrumented code.

3.6.2 Conflict With Clone

We have observed several false reports in the presence of the `clone` system call[2]. First, a process calls `clone` with the `CLONE_VM|CLONE_FILES` flags, which creates a child process that shares memory with the parent. In particular, the memory used by the child's stack still belongs to the parent. Then the child process calls a function that has objects on the stack and the AddressSanitizer instrumentation poisons the stack object redzones. Finally, without exiting the function and un-poisoning the redzones, the child process calls a function that never returns (e.g., `_exit` or `exec`). As a result, part of the parent address space remains poisoned and AddressSanitizer reports an error later when this memory is reused. We solved this problem by finding `never_return` function calls (functions like `_exit` or `exec` have this attribute) and unpoisoning the entire stack memory before the call. For

[2] http://code.google.com/p/address-sanitizer/issues/detail?id=37

similar reasons the AddressSanitizer run-time library has to intercept `longjmp` and C++ exceptions.

3.6.3 Intentional Wild Dereferences

We have seen several cases where a function intentionally reads *wild* memory locations. For example, low level code iterates between two addresses on the stack crossing multiple stack frames. For these cases we have implemented a function attribute `no_address_safety_analysis` which should be added to the function declaration in the C/C++ source. These cases are rare; e.g., in the Chromium browser we needed this attribute only once.

3.7 Threads

AddressSanitizer is thread-safe. The shadow memory is modified only when the corresponding application memory is not accessible (inside `malloc` or `free`, during creation or destruction of a stack frame, during module initialization). All other accesses to the shadow memory are reads. The `malloc` and `free` functions use thread-local caches to avoid locking on every call (as most modern `malloc` implementations do). If the original program has a race between a memory access and deletion of that memory, AddressSanitizer may sometimes detect it as a use-after-free bug, but is not guaranteed to. Thread IDs are recorded for every `malloc` and `free` and are reported in error messages together with thread creation call stacks.

4 Evaluation

We measured the performance of AddressSanitizer on C and C++ benchmarks from SPEC CPU2006 [28]. The measurements were done in 64-bit mode on an HP Z600 machine with 2 quad-core Intel Xeon E5620 CPUs and 24GB RAM. We compared the performance of instrumented binaries with the binaries built using the regular LLVM compiler (`clang -O2`). We used 32-byte redzones, disabled the stack unwinding during `malloc` and `free` and set the quarantine size to zero (see Section 4.3).

Figure 2 shows that the average slowdown on CPU2006 is 73%. The largest slowdown is seen on perlbench and xalancbmk (2.60x and 2.67x respectively). These two benchmarks are very `malloc`-intensive and make a huge number of 1- and 2- byte memory accesses (both benchmarks are text processing programs). We also measured AddressSanitizer performance when only writes are instrumented: the average slowdown is 26%. This mode could be used in performance-critical environments to find a subset of memory bugs.

Table 1: Memory usage with AddressSanitizer (MB)

Benchmark	Original	Instrumented	Increase
400.perlbench	670	2168	3.64x
401.bzip2	858	1618	2.12x
403.gcc	893	4133	5.21x
429.mcf	1684	2098	1.40x
445.gobmk	37	369	11.22x
456.hmmer	33	582	19.84x
458.sjeng	180	249	1.56x
462.libquantum	104	930	10.06x
464.h264ref	72	439	6.86x
471.omnetpp	181	787	4.89x
473.astar	343	1214	3.98x
483.xalancbmk	434	1688	4.38x
433.milc	694	1618	2.62x
444.namd	58	146	2.83x
447.dealII	807	2602	3.63x
450.soplex	637	2479	4.38x
453.povray	17	371	24.55x
470.lbm	417	550	1.48x
482.sphinx3	52	426	9.22x
total	8171	24467	**3.37**x

Three bugs were found in CPU2006: one stack and one global buffer overflow in h264ref, and a use-after-realloc in perlbench.

We also evaluated the performance of different mapping `Scale` and `Offset` values (see Section 3.1). The `Scale` values greater than 3 produce slightly slower code on average (from 2% speedup to 15% slowdown compared to `Scale=3`). The memory footprint for `Scale=4,5` is close to that of `Scale=3`. For values 6 and 7, the memory footprint is greater because larger redzones are required. Setting `Offset` to zero (which requires `-fPIE/-pie`) gives a small speedup, bringing the average slowdown on CPU2006 to 69%.

Table 1 summarizes the increase in memory usage (collected by reading the VmPeak field from `/proc/self/status` at process termination). The memory overhead comes mostly from the `malloc` redzones. The average memory usage increase is 3.37x. There is also a constant-size overhead for quarantine, which we did not count in the experiment.

Table 2 summarizes the stack size increase (`VmStk` field in `/proc/self/status`). Only 6 benchmarks had a noticeable stack size change and only 3 benchmarks had stack size increases over 10%.

The binary size increase on SPEC CPU2006 ranges from 1.5x to 3.2x, with an average of 2.5x.

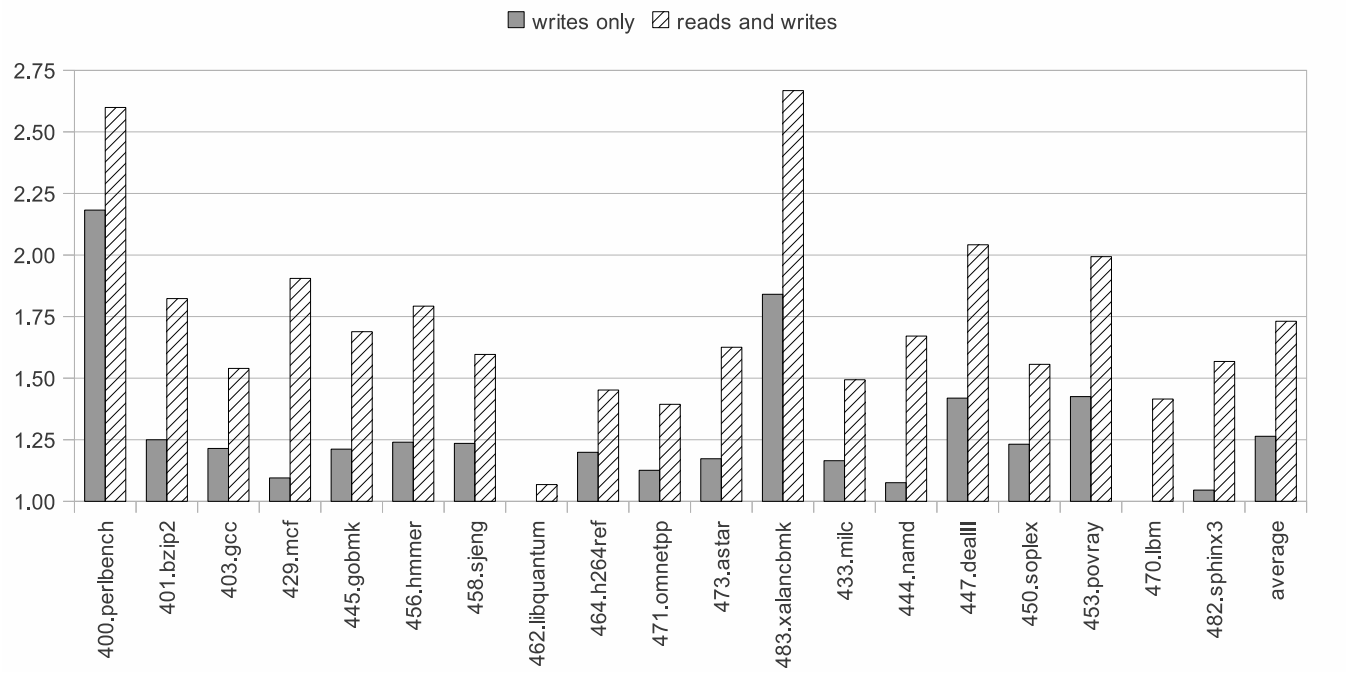

Figure 2: The average slowdown on SPEC CPU2006 on 64-bit Linux.

Table 2: Stack increase with AddressSanitizer (KB)

Benchmark	Original	Instrumented	Increase
400.perlbench	568	1740	3.06x
445.gobmk	184	264	1.43x
458.sjeng	828	848	1.02x
483.xalancbmk	2116	4720	2.23x
453.povray	88	96	1.09x
482.sphinx3	248	252	1.02x

4.1 Comparison

Comparing AddressSanitizer with other tools is tricky because other tools find different sets of bugs. Valgrind and Dr. Memory incur 20x and 10x slowdowns on CPU2006, respectively [8]. But these tools detect a different set of bugs (in addition to out-of-bounds and use-after-free they detect uninitialized reads and memory leaks, but do not handle out-of-bounds for most stack variables and globals).

Mudflap, probably the most similar tool to AddressSanitizer, has very unusual performance characteristics. According to our measurements, Mudflap's slowdown on CPU2006 ranges from 2x to 41x; several benchmarks fail with out-of-memory errors.

Debug malloc implementations that use CPU guard pages typically slow down only very `malloc`-intensive applications. Duma, the freely available guard page implementation for Linux, crashed on 12 out of 18 CPU2006 benchmarks with out-of-memory errors. No surprise: the Duma manual describes it as a "terrible memory hog". On the remaining 6 benchmarks it showed very small overhead (from -1% to 5%).

The overhead of the DieHarder debug malloc is very low, 20% on average [22]. However, on three `malloc`-intensive benchmarks it is comparable with the overhead of AddressSanitizer: perlbench, 2x; omnetpp, 1.85x; xalancbmk, 1.75x.

4.2 AddressSanitizer Deployment

The Chromium open-source browser [1] has been regularly tested with AddressSanitizer since we released the tool in May 2011. In the first 10 months of testing the tool detected over 300 previously unknown bugs in the Chromium code and in third-party libraries. 210 bugs were *heap-use-after-free*, 73 were *heap-buffer-overflow*, 8 *global-buffer-overflow*, 7 *stack-buffer-overflow* and 1 `memcpy` parameter overlap. In 13 more cases AddressSanitizer triggered some other kind of program error (e.g., uninitialized memory read), but did not provide a meaningful error message.

The two major sources of bug reports in Chromium are regular runs of the existing unit tests and targeted random test generation (fuzzing). In either case the speed of the instrumented code is critical. For unit tests, the high speed allows the use of fewer machines to keep up with the source changes. For fuzzing, it allows running

a random test in just a few seconds (since AddressSanitizer is implemented as compile-time instrumentation, it has no start-up penalty), and once a bug is found, minimize the test in reasonable time. A small number of bugs have been found by manually running the instrumented browser — which would not have been possible with a significantly slower tool.

Besides Chromium we have also tested a large amount of other code and found many bugs. As in Chromium, heap-use-after-free was the most frequent kind of bug; however stack- and global-buffer-overflow appeared more often than in Chromium. Several heap-use-after-free bugs were detected in LLVM itself. We were notified about bugs found by AddressSanitizer in Firefox, Perl, Vim, and several other opensource projects[3].

4.3 Tuning Accuracy And Resource Usage

AddressSanitizer has three major controls that affect accuracy and resource usage.

Depth of stack unwinding (default: 30). On every call to `malloc` and `free` the tool needs to unwind the call stack so that error messages contain more information. This option affects the speed of the tool, especially if the tested application is `malloc`-intensive. It does not affect the memory footprint or the bug-finding ability, but short stack traces are often not enough to analyze an error message.

Quarantine size (default: 256MB). This value controls the ability to find heap-use-after-free bugs (see Section 3.5). It does not affect performance.

Size of the heap redzone (default: 128 bytes). This option affects the ability to find heap-buffer-overflow bugs (see Section 3.5). Large values may lead to significant slowdown and increased memory usage, especially if the tested program allocates many small chunks of heap memory. Since the redzone is used to store the `malloc` call stack, decreasing the redzone automatically decreases the maximal unwinding depth.

While testing Chromium we used the default values of these three parameters. Increasing any of them did not increase the bug-finding ability. While testing other software we sometimes had to use smaller redzone sizes (32 or 64 bytes) and/or completely disable stack unwinding to meet extreme memory and speed constraints. In environments with a small amount of RAM we used smaller quarantine sizes. All three values are controlled by an environment variable and can be set at process startup.

[3] http://code.google.com/p/address-sanitizer/wiki/FoundBugs

5 Future Work

This section discusses improvements and further steps that could be taken with AddressSanitizer.

5.1 Compile-time Optimizations

It is not necessary to instrument every memory access in order to find all memory bugs. There is redundant instrumentation that can be eliminated, as shown in this example:

```
void inc(int *a) {
  (*a)++;  }
```

Here we have two memory accesses, one load and one store, but we need to instrument only the first one. This is the only compile-time optimization implemented in AddressSanitizer currently. Some other possible optimizations are described below. These optimizations apply only under certain conditions (e.g., there should be no non-pure function calls between the two accesses in the first example).

• Instrument only the first access:

```
*a = ...
if (...)
  *a = ...
```

• Instrument only the second access (although this gives up the property of guaranteeing to report an error prior to the actual load or store taking place):

```
if (...)
  *a = ...
*a = ...
```

• Instrument only `a[0]` and `a[n-1]`:

```
for (int i = 0; i < n; i++)
  a[i] = ...;
```

We already use this approach for instrumenting functions like `memset`, `memcpy` and similar. It may potentially miss some bugs if `n` is large.

• Combine two accesses into one:

```
struct { int a, b; } x;  ...
x.a = ...;
x.b = ...;
```

• Do not instrument accesses that can be statically proven to be correct:

```
int x[100];
for (int i = 0; i < 100; i++)
  x[i] = ...;
```

• No point in instrumenting accesses to scalar globals:

```
int glob;
int get_glob() {
  return glob; }
```

5.2 Handling Libraries

The current implementation of AddressSanitizer is based on compile-time instrumentation and thus does not handle system libraries (it does, however, handle some C library functions such as `memset`). For the open source libraries the best approach might be to create special instrumented builds. For the closed source libraries a combined static/dynamic instrumentation approach could be used. All available source code could be built with an AddressSanitizer-enabled compiler. Then, during execution, the closed-source libraries could be instrumented with a binary translation system (such as DynamoRIO [7, 6]).

It is possible to implement AddressSanitizer using only run-time instrumentation but it will probably be slower due to binary translation overhead, including suboptimal register allocation. Besides, it is not clear how to implement redzones for stack objects using run-time instrumentation.

5.3 Hardware Support

The performance characteristics of AddressSanitizer allow for use in a wide range of situations. However, for the most performance-critical applications and for the cases where the binary size is important, the current overhead may be too restrictive. The instrumentation performed by AddressSanitizer (see Section 3.2) could be replaced by a single new hardware instruction `checkN` (e.g., "`check4 Addr`" for a 4-byte access). The `checkN` instruction with parameter `Addr` should be equivalent to

```
ShadowAddr = (Addr >> Scale) + Offset;
k = *ShadowAddr;
if (k != 0 && ((Addr & 7) + N > k)
    GenerateException();
```

The values of `Offset` and `Scale` could be stored in special registers and set at application startup.

Such an instruction would improve performance by reducing the icache pressure, combining simple arithmetic operations, and achieving better branch prediction. It would also reduce the binary size significantly.

By default the `checkN` instruction could be a no-op and only enabled by a special CPU flag. This would allow to selectively test certain executions or even test long-lived processes for a fraction of their execution time.

6 Conclusions

In this paper we presented AddressSanitizer, a fast memory error detector. AddressSanitizer finds out-of-bounds (for heap, stack, and globals) accesses and use-after-free bugs at the cost of 73% slowdown on average; the tool has no false positives.

AddressSanitizer uses shadow memory to provide accurate and immediate bug detection. The conventional wisdom is that shadow memory either incurs high overhead through multi-level mapping schemes or imposes prohibitive address space requirements by occupying a large contiguous region. Our novel shadow state encoding reduces our shadow space footprint enough that we can use a simple mapping, which can be implemented with low overhead.

The high speed provided by the tool allows the user to run more tests faster. The tool has been used to test the Chromium browser and has found over 300 real bugs in just 10 months, including some that could have potentially led to security vulnerabilities. AddressSanitizer users found bugs in Firefox, Perl, Vim and LLVM.

The instrumentation required for AddressSanitizer is simple enough to be implemented in a wide range of compilers, binary instrumentation systems, and even in hardware.

Availability

AddressSanitizer is open source and is integrated with the LLVM compiler tool chain [4] starting from version 3.1. The documentation can be found at `http://clang.llvm.org/docs/AddressSanitizer.html`.

A Appendix: Instrumentation Examples

Here we give two examples of instrumentation on x86_64 (8- and 4- byte stores). C program:

```
void foo(T *a) {
  *a = 0x1234;
}
```

8-byte store:

```
clang -O2 -faddress-sanitizer a.c -c -DT=long

push   %rax
mov    %rdi,%rax
shr    $0x3,%rax
mov    $0x100000000000,%rcx
or     %rax,%rcx
cmpb   $0x0,(%rcx) # Compare Shadow with 0
jne    23 <foo+0x23>  # To Error
movq   $0x1234,(%rdi) # Original store
pop    %rax
retq
callq  __asan_report_store8 # Error
```

4-byte store:

```
clang -O2 -faddress-sanitizer a.c -c -DT=int
```

```
push   %rax
mov    %rdi,%rax
shr    $0x3,%rax
mov    $0x100000000000,%rcx
or     %rax,%rcx
mov    (%rcx),%al  # Get Shadow
test   %al,%al
je     27 <foo+0x27> # To original store
mov    %edi,%ecx  # Slow path
and    $0x7,%ecx  # Slow path
add    $0x3,%ecx  # Slow path
cmp    %al,%cl
jge    2f <foo+0x2f> # To Error
movl   $0x1234,(%rdi) # Original store
pop    %rax
retq
callq  __asan_report_store4  # Error
```

References

[1] The Chromium project. http://dev.chromium.org.

[2] Dmalloc – Debug Malloc Library. http://www.dmalloc.com.

[3] D.U.M.A. – Detect Unintended Memory Access. http://duma.sourceforge.net/.

[4] The LLVM Compiler Infrastructure. http://llvm.org.

[5] Emery D. Berger and Benjamin G. Zorn. DieHard: probabilistic memory safety for unsafe languages. In *PLDI 06*, pages 158–168. ACM Press, 2006.

[6] Derek Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, M.I.T., September 2004.

[7] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proc. of the International Symposium on Code Generation and Optimization (CGO '03)*, pages 265–275, March 2003.

[8] Derek Bruening and Qin Zhao. Practical memory checking with Dr. Memory. In *Proc. of the International Symposium on Code Generation and Optimization (CGO '11)*, pages 213–223, April 2011.

[9] Marc Brünink, Martin Süßkraut, and Christof Fetzer. Boundless memory allocations for memory safety and high availability. In *Proc. of the 41st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2011)*. IEEE Computer Society, June 2011.

[10] Winnie Cheng, Qin Zhao, Bei Yu, and Scott Hiroshige. Tainttrace: Efficient flow tracing with dynamic binary rewriting. In *Proc. of the 11th IEEE Symposium on Computers and Communications (ISCC '06)*, pages 749–754, 2006.

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

[12] Niranjan Hasabnis, Ashish Misra, and R. Sekar. Light-weight bounds checking. In *Proc. of the International Symposium on Code Generation and Optimization (CGO '12)*, pages 135–144, April 2012.

[13] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *Proc. of the Winter USENIX Conference*, pages 125–136, January 1992.

[14] IBM Research. GCC extension for protecting applications from stack-smashing attacks. http://researchweb.watson.ibm.com/trl/projects/security/ssp/.

[15] Intel. Intel Parallel Inspector. http://software.intel.com/en-us/intel-parallel-inspector/.

[16] Mac OS X Developer Library. Memory Usage Performance Guidelines: Enabling the Malloc Debugging Features. http://developer.apple.com/library/mac/#documentation/darwin/reference/manpages/man3/libgmalloc.3.html.

[17] Micro Focus. BoundsChecker. http://www.microfocus.com/products/micro-focus-developer/devpartner/visual-c.aspx.

[18] Microsoft Support. How to use Pageheap.exe in Windows XP, Windows 2000, and Windows Server 2003. http://support.microsoft.com/kb/286470.

[19] George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-safe retrotting of legacy code. In *Proc. of the , Principles of Programming Languages*, pages 128–139, 2002.

[20] Nicholas Nethercote and Julian 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)*, pages 65–74, June 2007.

[21] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*, pages 89–100, June 2007.

[22] Gene Novark and Emery D. Berger. DieHarder: securing the heap. In *Proc. of the 17th ACM conference on Computer and communications security*, CCS '10, pages 573–584. ACM, 2010.

[23] Oracle. Sun Memory Error Discovery Tool (Discover). http://download.oracle.com/docs/cd/E19205-01/821-1784/6nmoc18gq/index.html.

[24] Parasoft. Insure++. http://www.parasoft.com/jsp/products/insure.jsp?itemId=63.

[25] Bruce Perens. Electric Fence. http://perens.com/FreeSoftware/ElectricFence/.

[26] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. LIFT: A low-overhead practical information flow tracking system for detecting security attacks. In *Proc. of the 39th International Symposium on Microarchitecture (MICRO 39)*, pages 135–148, 2006.

[27] Julian Seward and Nicholas Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *Proc. of the USENIX Annual Technical Conference*, pages 2–2, 2005.

[28] Standard Performance Evaluation Corporation. SPEC CPU2006 benchmark suite, 2006. http://www.spec.org/osg/cpu2006/.

[29] Perry Wagle and Crispin Cowan. Stackguard: Simple stack smash protection for gcc. In *Proc. of the GCC Developers Summit*, pages 243–255, 2003.

[30] Qin Zhao, Derek Bruening, and Saman Amarasinghe. Efficient memory shadowing for 64-bit architectures. In *Proc. of the The International Symposium on Memory Management (ISMM '10)*, pages 93–102, Jun 2010.

[31] Qin Zhao, Derek Bruening, and Saman Amarasinghe. Umbra: Efficient and scalable memory shadowing. In *Proc. of the International Symposium on Code Generation and Optimization (CGO '10)*, pages 22–31, April 2010.