The comparison between AddressSanitizer and Valgrind memcheck

2019 Spring Quarter

Secure Software Engineering

Daniel Felix Ritchie School of
Engineering & Computer Science

Daniel Felix Ritchie School of Engineering & Computer Science
University of Denver, 2155 E. Wesley Ave.
Denver, CO 80210, 303-871-3787
Hao.Wu81@du.edu
917-499-3326

Introduction

Memory access bugs, including buffer overflows and uses of freed heap memory (Double free), is a serious problem for programming languages like C and C++. There are two different good tools to detect these problems. One is AddressSanitizer from Google, another is Valgrind memcheck.

AddressSanitizer contains two parts: an instrumentation module and a run-time library. The instrumentation module modifies the code which is to detect to check shadow memory state for each memory access and creates poisoned red-zones around stack and global objects to detect overflows and underflows. The current implementation is based on the LLVM compiler infrastructure. The run-time library replaces malloc, free and related functions, creates poisoned red zones around allocated heap regions, delays the reuse of freed heap regions, and report errors.

Valgrind Memcheck, a practical tool which has a similar function of AddressSanitizer. And Memcheck is implemented with the dynamic binary instrumentation framework Valgrind. So the different is that it checks programs for errors as they run. Memcheck performs four kinds of memory error checking. First, it tracks the address of every byte of memory, updating the information if memory is allocated and freed. With this information, it can detect all accesses to unaddressable memory. Second, it tracks memory allocated with malloc(), new and new[]. So it can detect fault or repeated frees of heap blocks, and can detect memory leaks at program termination. Third, it checks that memory blocks of functions like strcpy() and memcpy(). Fourth, it tracks the definedness of every bit of data in registers and memory. So it can detect undefined value errors with bit-precision.

About the methods used, AddressSanitizer is based on directly-mapped shadow memory detecting, and Memcheck is based on dynamic binary instrumentation framework (different kind of shadow memory to AddressSanitizer). AddressSanitizer is created when code is compiled and inserts error-checking code inline into a program. Its shadow-memory implementation uses a huge reservation of virtual memory for its shadow memory, giving very different performance characteristics. Memcheck runs outside the program.

About the function, AddressSanitizer and Memcheck both detect accesses to unaddressable memory, memory leaks at program termination, memory allocation function. But AddressSanitizer can additionly detect out-of-bounds bugs in the stack. Memcheck can additionally find uninitialized reads. AddressSanitizer only has cost of 73% slow down and 3.4x increased memory usage. Memcheck programs typically run 20 to 30 times slower than normal, it is obviously slower than AddressSanitizer. Memcheck can be used to any program language because it instruments and analyses executable machine code, rather than source code or object code. But AddressSanitizer is only good at C and C++.

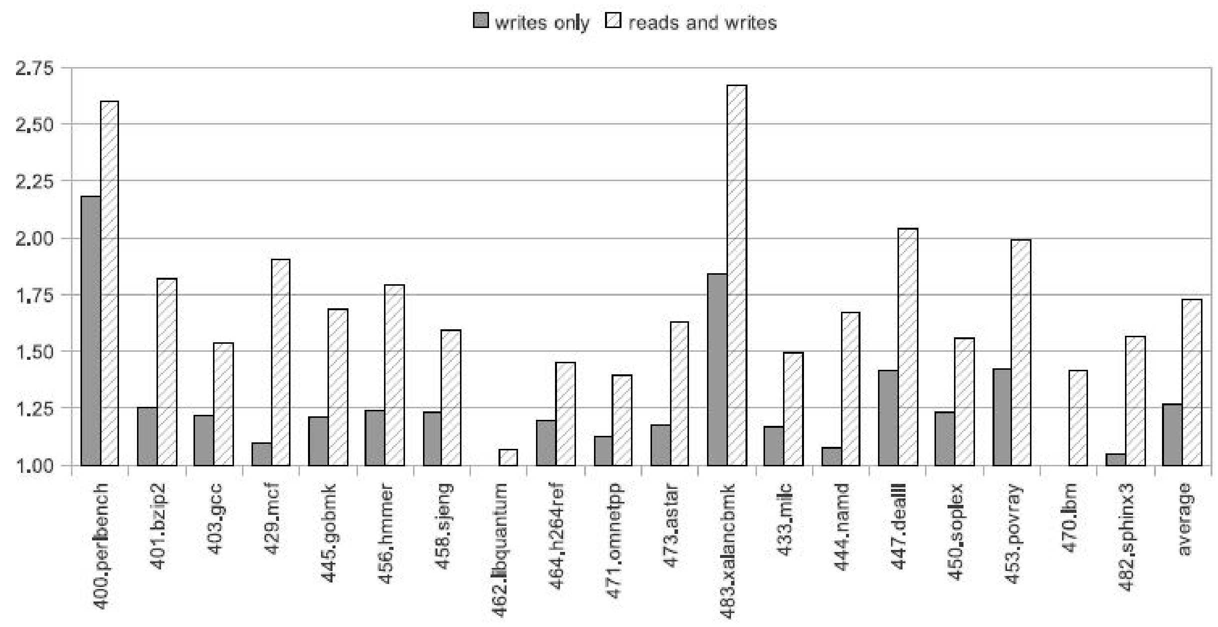
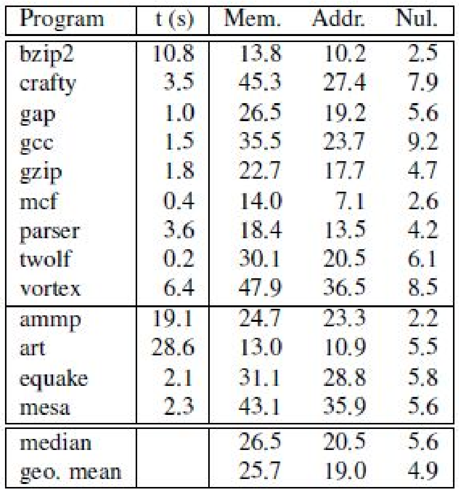


Figure 2: The average slowdown on SPEC CPU2006 on 64-bit Linux.



Program	t (s)	Mem.	Addr.	Nul.
bzip2	10.8	13.8	10.2	2.5
crafty	3.5	45.3	27.4	7.9
gap	1.0	26.5	19.2	5.6
gcc	1.5	35.5	23.7	9.2
gzip	1.8	22.7	17.7	4.7
mcf	0.4	14.0	7.1	2.6
parser	3.6	18.4	13.5	4.2
twolf	0.2	30.1	20.5	6.1
vortex	6.4	47.9	36.5	8.5
ammp	19.1	24.7	23.3	2.2
art	28.6	13.0	10.9	5.5
equake	2.1	31.1	28.8	5.8
mesa	2.3	43.1	35.9	5.6
median		26.5	20.5	5.6
geo. mean		25.7	19.0	4.9

Different Shadow memory technology

Shadow memory is a technique which is used to track and store information on computer memory during its execution. Shadow memory consists of shadow bytes that map to individual bits or one or more bytes in main memory. These shadow bytes are typically invisible to the original program and are used to record information about the original piece of data.

Memcheck uses shadow memory to store metadata corresponding to each piece of application data. An application address is mapped to a shadow address by a direct scale and offset. And full application address space is mapped to a single shadow address space. But it uses multi-level translation schemes to provide more flexibility in address space layout.

On a high level, AddressSanitizer approach to memory error detection uses 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. And 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 Memcheck. As Figure 1 shown, this is a address space layout. The application memory is split into two parts (low and high) which map to the corresponding shadow regions. Original shadow address gives themselves' addresses in the Bad region, which is marked inaccessible with page protection.

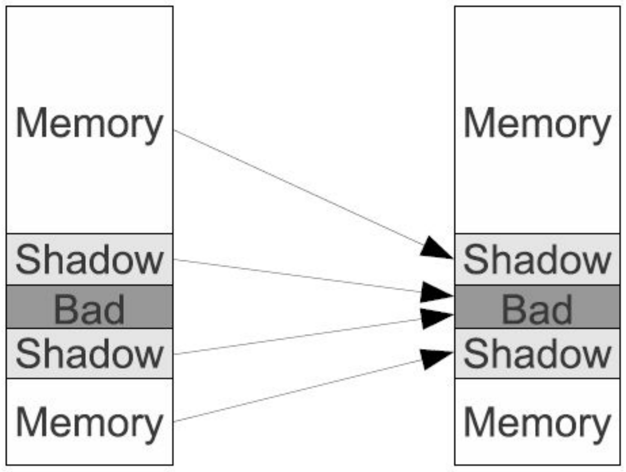


Figure 1: AddressSanitizer memory mapping.

Algorithm comparison

Instrumentation

For AddressSanitizer, it computes the address of the corresponding shadow byte, loads that byte, and checks whether it is zero when instrumenting 8-byte memory access. As below:

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

When instrumenting 1, 2 or 4-byte accesses, AddressSanitizer has a little complex method to instrumentation, if the shadow value is positive (For example, only the first k bytes in the 8-byte word are addressable), it compares the 3 last bits of the address with k. As below:

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

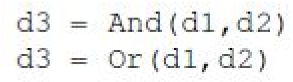
AddressSanitizer instrumentation inserts only one memory read for each memory access in the original code. This operation assumes that an N-byte access is aligned to N. So AddressSanitizer may miss a bug caused by an unaligned access. As below shown:

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

For Memcheck, it is based on V bit instrumentation scheme. A V bit of zero indicates that the corresponding data bit has a properly defined value, and a V bit of one indicates that it does not. It is weird, but makes some of the shadow operations more efficient than opposite if the bit values were inverted. For describing, d1 and d2 to denote virtual registers holding real values, and v1 and v2 denote virtual shadow registers holding V bits. It uses the following binary operations. 1. **DifD(v1,v2)** (defined if either defined) returns a V bit vector the same width as v1 and v2. 2. **UifU(v1,v2)** (undefined if either undefined) can be implemented using a bitwise-OR operation. 3. **ImproveAND(d,v) and ImproveOR(d,v)** are helpers for dealing with AND and OR operations. Every operation (instruction or system call) that creates a value must be instrumented by a shadow operation that computes the corresponding V bits. Instrumentation includes: 1. **Register and memory initialisation.** At startup, all registers' V bits are set to one (undefined). Except stack pointer, whose V bits is set to zero (defined). All memory bytes mapped at startup have their V bits set to zero (defined). **2. Memory allocation and deallocation.** The Valgrind framework intercepts function and system calls which causes usable address ranges to appear/disappear. Memcheck is notified of such events and marks shadow memory appropriately. 3. **Data copying operations.** Register-to-register moves cause a move between the corresponding shadow registers. Register-to-memory and memory-to-register transfers are instrumented with a move between the corresponding shadow register and

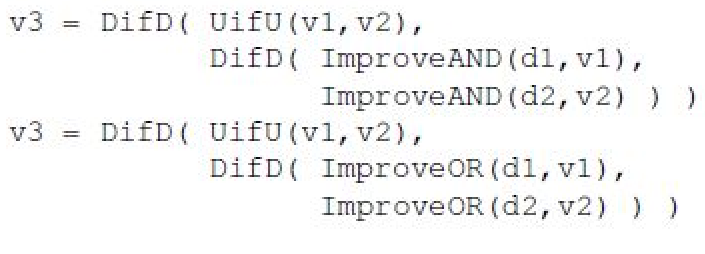
shadow memory. 4. **Addition and subtraction.** Given d3 = Add(d1,d2) each result bit can simplistically be considered defined if both the corresponding argument bits are defined. But a result bits could also be undefined due to an undefined carry/borrow propagating upwards from less significant bit positions. Therefore Memcheck needs to generate v3 = Left(UifU(v1,v2)) to check if either undefined. The same scheme is used for multiplies. This is too conservative because the product of two numbers with N and M consecutive least-significant defined bits

has N + M least-significant defined bits, rather than min(N;M) as the Add/Sub scheme generates. 5. **Xor.** For example, given d3 = Xor(d1,d2), generate v3 = UifU(v1,v2). The rule for Not is trivially derived by constant-folding the rule for Xor(0xFF..FF,d) giving, so the simple result v, where v is the shadow for d and unchanged. **6. And and Or.** These need to be checked the actual operand values and their shadow bits. As below shown, We have:



```
d3 = And(d1,d2)
d3 = Or(d1,d2)
```

the resulting instrumentation should be:



```
v3 = DifD( UifU(v1,v2),
           DifD( ImproveAND(d1,v1),
                 ImproveAND(d2,v2) ) )
v3 = DifD( UifU(v1,v2),
           DifD( ImproveOR(d1,v1),
                 ImproveOR(d2,v2) ) )
```

7. **Shl, Shr, Sar, Rol, Ror (Shift left,**

Unsigned shift right, Signed shift right, Rotate left, Rotate right). In these cases, if the shift/rotate amount is undefined, the entire result is undefined. Otherwise, the result V bits are got by applying the same shift/rotate operation to the V bits of the value which is shifted/rotated. 8. **Floating point (FP) and MMX, SSE, SSE2 (SIMD) operations.** Valgrind does not check floating point or SIMD instructions to the same level of detail like it does integer instructions. Instead, it only modifies some of the register fields in the instruction, marks any instructions which references memory, and copies them into the output instruction stream. Because of neither the FP nor SIMD registers have any associated V bits. When a value is loaded from memory into such a register, if any part of the value is undefined, an error message is issued. When a value is written from such a register to memory, shadow memory is marked as defined.

Issue error message

At every point where an undefined value could be consumed by an operation, We have a choice: should it report the error right now, or should it silently propagate the undefinedness into the result? Both approaches have advantages. **Reporting the error sooner (the eager strategy mentioned above)** makes it easier for users to track the root cause of undefined values. **Deferring error checking and reporting** has two advantages. Firstly, error checks are expensive. So minimising them improves performance. Secondly, reporting errors too soon can lead to false positives: undefined values might be used in a safe way and then deleted, so an early check on them would give a useless error report to the user.

The AddressSanitizer instrumentation is established at the end of the LLVM optimization pipeline. This method only detects those memory accesses which survived all scalar and loop optimizations performed by the LLVM optimizer. For example, memory accesses to local stack objects that are optimized by LLVM will not be instrumented. And we don’t have to instrument memory accesses generated by the LLVM code generator. The error reporting code (ReportAndCrash(Addr)) is executed at most once, but it is inserted in many places in the code. AddressSanitizer uses a simple function call to achieve it.

Memcheck mostly takes the second method, deferring error reporting as far as it can, similar to AddressSanitizer. It only report undefined values when the program is in immediate danger of performing one of the following actions. But it could change its observable behaviour. 1. Taking a memory exception due to use of an unde-ned address in a load or store. 2. Making a conditional jump based on undefined condition codes. 3. Passing undefined values to a system call. 4. Loading uninitialised values from memory into a SIMD or FP register.

False Negatives

For AddressSanitizer, as we described that unaligned access bug. It ignores this type of bug right now because all solutions will slow down the common path. Possible solutions include: 1. check at run-time whether the address is unaligned. 2. use a byte-to-byte shadow mapping but it is only on 64-bit system. 3. use a more compact mapping to minimize the probability of missing this bug. AddressSanitizer may also miss bugs in other two cases. Firstly, if an out-of-bounds access touches memory which is too far away from the object bound, it may be seperated in a different valid allocation and this bug will be missed. As the example below:

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

Secondly, if a large amount of memory has been allocated and deallocated, the use-after-free may not be detected. As the example below:

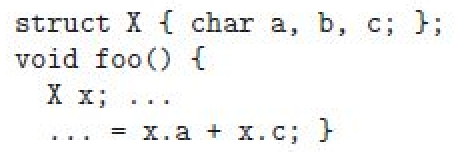
```
char *a = new char[1 << 20];  // 1MB
delete [] a;  // <<< "free"
char *b = new char[1 << 28];  // 256MB
delete [] b;  // drains the quarantine queue.
char *c = new char[1 << 20];  // 1MB
a[0] = 0;     // "use". May land in 'c'.
```

For Memcheck, The false negatives includes: 1. Caller-saved registers in procedures. it is possible that registers which should be regarded as undefined at the start of a callee are marked as defined due to previous activity in the caller, and so bugs might be missed. 2. Programs that switch stacks (Implement user-space threading). There is no perfect way to distinguish a large stack allocation or deallocation from a stack-switch. (Similar to AddressSanitizer bug 3) Valgrind uses a method: any change in the stack pointer greater than 2MB will be assumed to be a stack-switch. When Valgrind judges that a stack-switch has happened, Memcheck won't take any further actions. So if a stack frame above 2MB is allocated, Valgrind considers this as a stack switch, and Memcheck will not mark the newly allocated area as undefined. The program could then use the values in the unsafe state, and Memcheck will not detect this bug. In addition, Memcheck cannot detect errors on code paths that are not executed, nor can it detect errors

arising from unseen combinations of inputs. This is an inherent problem of dynamic analysis.

False positives

As Google said, AddressSanitizer has no false positives. But during the AddressSanitizer development and deployment they have seen a number of undesirable error reports that has been fixed includes: 1. **Conflict With Load Widening.** As below code shown:



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

The object x has size 3 and alignment 4. Load widening transforms x.a+x.c into a 4-byte load, which partially crosses the object boundary. Then AddressSanitizer will instrument this 4-byte load which leads to a false positive in the optimization pipeline. Google partially disabled load widening in LLVM when AddressSanitizer instrumentation is ran. It still permit widening x.a+x.b into a 2-byte load, because this transformation will not cause this problem. **2. Conflict With Clone.** Firstly, a process calls which is cloned by the CLONE VM|CLONE FILES flags creates a child process that shares memory with the parent. When the memory used by the child's stack still belongs to the parent. The child process will call a function that has objects on the stack and the AddressSanitizer instrumentation poisons the stack object redzones. Finally, without exiting the function and unpoisoning the red-zones, the child process calls a function that never returns (exit or exec). The problem is that part of the parent address space remains poisoned and AddressSanitizer reports an error later when this memory is reused. Google finds never return function calls and unpoisoning the entire stack memory before the call and solve this bug.

For Memcheck, a few hand-coded assembly sequences, and a few rare compiler-generated usage may cause false positives. **1. xor %reg,%reg: %reg.** It is defined after the instruction. This is solved by Valgrind x86-to-UCode translation phase, which translates this code to mov $0,%reg; xor; %reg, %reg. **2. sbb %reg,%reg.** This copies the carry flag into all bits of %reg, and does not depend on %reg's original value. This instrumentation saved any undefinedness. This is solved by instead translating **mov $0,%reg; sbb %reg,%reg. 3. GNU libc** has highly-optimised, hand-written assembly routines for common string functions, like strlen(). These traverse the string at one time. For such code, Memcheck only uses the Left operator to model this and leads to false positives. **4. Memcheck's underlying assumptions** are sometimes invalid. For example, some programs use undefined values as an additional source of entropy when generating random numbers.

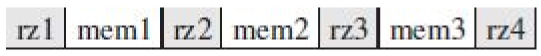
Other characteristic

Thread of AddressSanitizer

AddressSanitizer is thread-safe. The shadow memory will be modified only when the corresponding application memory would not be accessible. All other accesses to the shadow memory are reading. The malloc and free functions use thread-local caches to avoid locking on every call. If the original program has a race between a memory access and deletion of that memory, AddressSanitizer may detect it as a use-after-free bug. Thread IDs are recorded for every malloc and free and are reported in error messages together with thread creation call stacks.

Run-time Library of AddressSanitizer

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. And the Bad segment of the shadow memory will be protected. The memory regions inside the allocator are assigned 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 red zones will be allocated from the operating system. For n regions AddressSanitizer allocate n+1 red zones, so that the right redzone of one region is a left redzone of another region as below shown,



The left redzone is used to store the internal data of the allocator. So the minimum size of the heap redzone is 32 bytes.

Reference

[1] Julian Seward and Nicholas Nethercote, *Using Valgrind to detect undefined value errors with bit-precision.* In Proceedings of the USENIX'05 Annual Technical Conference, Anaheim, California, USA, April 2005.

[2] Nicholas Nethercote and Julian Seward. *Valgrind: A program supervision framework*. In Proceedings of RV'03, Boulder, Colorado, USA, July 2003.

[3] Nicholas Nethercote and Julian Seward, *How to Shadow Every Byte of Memory Used by a Program.* In Proceedings of the Third International ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE 2007), San Diego, California, USA, June 2007.

[4] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, Dmitry Vyukov, *AddressSanitizer: A Fast Address Sanity Checker.* In Proceedings of the USENIX ATC 2012. Google, 2012.

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

[6] Wikipedia. *"Shadow Memory."* Wikipedia. March 29, 2019. Accessed June 04, 2019. https://en.wikipedia.org/wiki/Shadow_memory