MemorySanitizer: fast detector of uninitialized memory use in C++

Evgeniy Stepanov
Google
eugenis@google.com

Konstantin Serebryany
Google
kcc@google.com

Abstract

This paper presents MemorySanitizer, a dynamic tool that detects uses of uninitialized memory in C and C++. The tool is based on compile time instrumentation and relies on bit-precise shadow memory at run-time. Shadow propagation technique is used to avoid false positive reports on copying of uninitialized memory.

MemorySanitizer finds bugs at a modest cost of 2.5x in execution time and 2x in memory usage; the tool has an optional origin tracking mode that provides better reports with moderate extra overhead. The reports with origins are more detailed compared to reports from other similar tools; such reports contain names of local variables and the entire history of the uninitialized memory including intermediate stores. In this paper we share our experience in deploying the tool at a large scale and demonstrate the benefits of compile-time instrumentation over dynamic binary instrumentation.

1. Introduction

Unlike most other programming languages, C and C++ do not trade performance for safety. As a result, all stack and heap objects in C/C++ are created uninitialized (except when `calloc` is used). Consequently, *use of uninitialized memory* (*UUM*) remains a serious concern for C/C++ developers. UUMs are hard to find during testing as they do not necessary lead to failures on every execution. UUMs may trigger a failure when a completely unrelated change is applied to the program, or when developers start using another compiler, OS, system library, or machine configuration.

The authors have been involved in a large scale deployment of several UUM detectors at Google and have seen thousands of UUM bugs. In the majority of cases the code owners fix the discovered bugs quickly, but they are often reluctant to use the tools themselves, mainly because of huge slowdown incurred by the existing tools. This slowdown consists of two parts: first, there is a start-up penalty from dynamic binary instrumentation and second, there is a steady state slowdown from executing the instrumentation code itself. When executing short running tests or tests with large code size, the start-up penalty may dominate and the overal slowdown could be over 100x. Large steady state slowdown is a problem as well since many network or GUI applications simply cannot be run correctly when slowed down by 20x or more. These considerations led us to the development of MemorySanitizer, a new UUM detector that has moderate steady state slowdown and no startup penalty. Unlike other popular UUM detectors MemorySanitizer uses static compile-time instrumentation — essentially it fulfills the same task as the other tools by solving a simpler problem.

1.1 Contributions

In this paper we:

- demonstrate that static compiler instrumentation can be used to detect uses of uninitialized memory (UUM);
- describe a UUM detector that is an order of magnitude faster than other state of the art alternatives;
- propose an enhanced way of tracking origins of uninitialized memory that allows to produce more informative error messages.

1.2 Outline

In section 2 we discuss related work; then in section 3 we describe the MemorySanitizer algorithm, including the instrumentation, run-time library and origin tracking. In section 4 we evaluate the tool and compare it to other tools, and also describe our experience with deploying MemorySanitizer in large scale projects. In section 5 we discuss future work and then conclude the paper.

2. Related work

One of the first successful and widely used tools for detecting UUMs is Memcheck [18], based on Valgrind binary translation system [15]. Memcheck simultaneously detects UUMs and addressability bugs (heap buffer overflow and use after free). It normally uses 2 bits of shadow memory per byte of application memory; the shadow for every byte has 4 states: addressable and initialized, not addressable, address-

able but uninitialized, addressable and partially initialized. If the byte is partially initialized then the tool maintains a second layer of shadow, this time with bit-to-bit mapping. Memcheck propagates shadow bits through most instructions in the program and reports UUMs only when uninitialized memory may affect the program's behavior. This shadow propagation allows to achieve near-zero false positive rate. Memcheck's slowdown is typically around 20x, but may be much larger on multi-threaded programs since Valgrind is single-threaded.

Dr. Memory [9], a tool based on DynamoRIO binary translation system [8], is similar to Memcheck in many ways. Using a more modern, optimized, and multi-threaded binary translation system allows Dr. Memory to be twice as fast, compared to Memcheck on average (more if the application is heavily threaded). However, a 2-bit-per-byte shadow combined with multi-threaded execution leads to subtle false positives caused by concurrent updates of adjacent shadow bits[1].

Memcheck and Dr. Memory combine the detection of UUMs and addressability bugs in a single tool. On the one hand, this clearly simplifies the usage. But on the other hand, the overheads required to detect UUMs and addressability bugs multiply. In particular, detection of addressability bugs requires red zones to find buffer overflows and delaying memory reuse to find use after free, but it needs only 1 bit of shadow per byte, or even less. Detecting UUMs requires fatter shadow memory (up to bit-per-bit in the worst case), but does not need red zones and quarantine. Combining UUM and addressability bug detection requires the use of fat shadow for red zones and quarantine – as the result the overheads in memory multiply. This is why Memcheck and Dr. Memory have to use compact 2-bit-per-byte shadow for the common case, even though it slows down the instrumented code and for Dr. Memory also leads to false positives. MemorySanitizer solves this problem by detecting only UUMs. A sibling tool based on similar principles, AddressSanitizer [17], finds addressability bugs without trying to find UUMs. As we show in Section 4.3, running MemorySanitizer followed by AddressSanitizer is still much faster than either Valgrind or Dr. Memory in most cases.

A feature that distinguishes Memcheck from Dr. Memory is its origin tracking (`--track-origins=yes`). Unlike most other kinds of bugs, such as e.g. buffer overflows, where something bad happens in a particular place of program, UUMs are harder to analyze because there is no bad event in the program – instead, the program lacks an event of memory initialization. Since Memcheck (correctly) does not report UUMs when copying uninitialized memory, the point of the actual report may be very far from the point of the memory allocation. Origin tracking allows the tool to report the particular memory region from which the uninitialized memory originated.

[1] https://code.google.com/p/drmemory/issues/detail?id=1557

Origin tracking described in the original paper [7] is based on *value piggybacking* technique, which stores origin information in the spare bits of an undefined value. This approach works well for propagating origins of null pointers in Java, but for UUM origins it has severe limitations: origin information can be destroyed in arithmetic operations and partial copies, and it is not applicable to values that are smaller than 32 bits. Current version of Memcheck implements a different and, to the best of our knowledge, unpublished approach of storing origin information in shadow memory.

MemorySanitizer implements origin tracking with shadow storage similar to Memcheck, described in more detail in Section 3.5. We go one step further and implement an "advanced" origin tracking mode (Section 3.6), which records all memory stores along the path from the allocation to the use of the uninitialized value.

Intel Inspector XE [10] (previously called Intel Parallel Inspector) is a commercial memory error detector based on PIN binary translation system [14]; Inspector's functionality is similar to that of Dr. Memory and Memcheck. The details of Inspector implementation are not public, however at least the version released in 2011 suffered from frequent false positives supposedly because the tool did not propagate shadow bits through memory copies[2]. The version released in 2013 seems to have fewer false positives, however it still incurs overhead of up to 500x, which significantly limits the tool usefulness.

All three tools are based on dynamic binary translation, which means that for short-running tests the translation overhead is significant, see Section 4.3. To the best of our knowledge, MemorySanitizer is the first tool for detecting UUMs in C and C++ based on static compiler instrumentation.

There are also tools that do not use code instrumentation to detect UUMs. DieHard [6] and DieHarder [16] attempt to find UUMs by initializing heap-allocated memory with specialized patterns. This approach is much simpler and faster, but it is probabilistic (i.e. does not guarantee the detection of UUM) and does not provide detailed error messages.

3. MemorySanitizer algorithm

This section describes MemorySanitizer algorithm: shadow memory, instrumentation, run-time support, and origin tracking.

3.1 Shadow memory

MemorySanitizer employs 1-to-1 shadow mapping, i.e. for each bit of application memory the tool keeps one bit of shadow memory. This approach allows very cheap computation of shadow address; in the current implementation, given the application memory address Addr, the corresponding shadow address is

[2] https://software.intel.com/en-us/forums/topic/267308

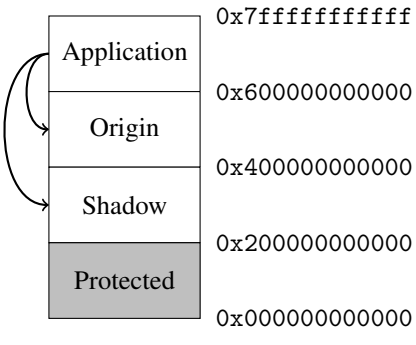

Figure 1. MemorySanitizer memory mapping

```
Addr & ShadowMask
```

The `ShadowMask` constant is platform-specific; for x86_64 Linux it is chosen as ~0x400000000000 to ensure that shadow memory occupies the normally unused 32Tb memory region at address 0x200000000000. See Figure 1.

Each bit of MemorySanitizer shadow encodes the state of the corresponding bit of the application memory, where value 0 stands for initialized, or *defined* bit, and value 1 — for uninitialized (*undefined*) bit.

For origin tracking (see Section 3.5), we allocate another region of the same size immediately following the shadow memory region. Both *Shadow* and *Origin* are mapped with MAP_NORESERVE flag, similarly to AddressSanitizer [17].

3.2 Shadow propagation

On a high level, our algorithm is similar to algorithms used in Dr. Memory and Memcheck, except that static compiler instrumentation is used instead of dynamic binary instrumentation.

All newly allocated memory is "poisoned", i.e. corresponding shadow memory is filled with 0xFF, signifying that all bits of this memory are uninitialized.

According to the C++ Standard [13, §4.1 p1], any lvalue-to-rvalue conversion on an uninitialized object has undefined behavior. C++14 relaxes this requirement for unsigned narrow character types [5, p1787]. Lvalue-to-rvalue conversion on an uninitialized object of such type produces an *indeterminate* value in C++14.

Commonly used compilers allow loading uninitialized values of integer and floating point types with expected results. Such operations are quite common in existing code, and must be allowed by any practical tool.

Another consideration is that MemorySanitizer is implemented as a compiler optimization pass. As such, it may observe memory loads that do not correspond to any reads in the program source. For example, for a class like this:

```
class A { char x; int y; };
```

a C++ compiler may generate a copying constructor that would simply do an 8-byte memory load including the potential uninitialized padding between class data members.

MemorySanitizer allows copying of uninitialized memory and a set of other "safe" operations with it without reporting an error. To handle such cases correctly, MemorySanitizer implements *shadow propagation*. Result of a load from uninitialized memory is an *undefined value*. This is modelled by assigning a *shadow value* to each compiler temporary, and storing that shadow value at the corresponding location in shadow memory when the application value is stored.

A subset of operations on application values require their operands to be initialized. These operations are, at minimum: conditional branch, system call and pointer dereference. An attempt to pass uninitialized memory as an argument of one of these instructions is reported as an error.

Most of the other operations propagate shadow by assigning a new shadow value to the operation result. The new shadow value is computed based on values of the operands and their shadow values. This computation depends on the operation, and is covered in more detail in section 3.3.

3.3 Instrumentation

MemorySanitizer instrumentation is implemented as an LLVM [3] optimization pass. Unlike AddressSanitizer and ThreadSanitizer [4] tools that only care about memory accesses, MemorySanitizer needs to handle all possible LLVM IR (SSA-based program representation) instructions either by checking operand shadow, or by propagating it to the result shadow.

For every IR temporary value MemorySanitizer creates another temporary that holds its shadow value. The type of the shadow value is determined as follows:

- The shadow value for a value of a scalar type (integer, floating point or any pointer) is an integer of the same bit length.
- The shadow value for a value of a SIMD vector type `<N x T>` is a vector of the same length whose elements are shadow values of the elements of the original vector, `<N x Shadow(T)>`.
- The shadow value for a value of an aggregate type is, recursively, an aggregate of shadow values of the original type members.

For example,

$$
\begin{aligned}
&Shadow(iN) && = iN\\
&Shadow(float) && = i32\\
&Shadow(double) && = i64\\
&Shadow(i8*) && = i64\\
&Shadow(<4\ x\ float>) && =<4\ x\ i32>\\
&Shadow(\{double, \{float, i1\}\}) && = \{i64, \{i32, i1\}\}
\end{aligned}
$$

Given an instruction $A = op\ B, C$, we generate one or more instructions implementing $A' = op'\ B, C, B', C'$, where A' stands for the shadow value corresponding to the application value A.

Table 1. Basic shadow propagation rules

$A = \texttt{load}\ P$	check P', $A' = \texttt{load}\ (P\ \&\ \text{ShadowMask})$
$\texttt{store}\ P,\ A$	check P', $\texttt{store}\ (P\ \&\ \text{ShadowMask}),\ A'$
$A = const$	$A' = 0$
$A = undef$	$A' = 0xff$
$A = B\ \&\ C$	$A' = (B'\ \&\ C')\ \|\ (B\ \&\ C')\ \|\ (B'\ \&\ C)$
$A = B\ \|\ C$	$A' = (B'\ \&\ C')\ \|\ (\sim B\ \&\ C')\ \|\ (B'\ \&\ \sim C)$
$A = B \operatorname{xor} C$	$A' = B'\ \|\ C'$
$A = B \ll C$	$A' = (\text{sign-extend}(C' \neq 0))\ \|\ (B' \ll C)$

Shadow propagation rules for several basic instructions are shown in Table 1.

3.3.1 Approximate propagation

It is not always possible to *efficiently* implement op' that correctly models op behaviour. Take integer addition, for example. A single undefined bit in the lower digit can affect any number of bits of the result due to a possible carry over. Modelling this behaviour would be difficult and prohibitively slow. Instead, we approximate shadow propagation in a way that does not result in false positives, and, at the same time, does not yield too many false negatives.

Approximate propagation must satisfy the following natural requirements:

- Zero shadow operands should produce zero shadow result.
- If one of the operands has a non-zero shadow bit in a location where corresponding value bit affects the operation result, then the result shadow must be non-zero.

Several kinds of instructions have stricter requirements. For example, bit shifts and bit logic operations are often used to extract individual field from bitfields. As adjacent fields may be not initialized, it is important that the result shadow matches the exact bits occupied by a particular field.

One convenient operation with these properties that is also very fast to compute is bitwise OR.

$$A = B + C \implies A' = B'\ |\ C'$$

3.3.2 Integer multiplication

Integer multiplication is tricky. If one of the operands has one or more zeroes in its least significant bits, the operation can be decomposed into a left shift followed by a multiplication by a smaller number.

$$A = B * (C * 2^D) \implies A = (B \ll D) * C$$

In this example, A would have zeroes in its D least significant bits. These bits are defined as long as D is defined.

It is important to note that a lot of these quirks never appear in the user code. Authors would argue that it is fine (and may be even desirable) to outright forbid multiplication of partially undefined values. Unfortunately, MemorySanitizer operates on LLVM IR level and not on source level. For performance reasons, MemorySanitizer runs near the end of the optimizer chain. The code that it works with has undergone heavy transformation and contains constructs that were not present in the source code. This puts additional requirements on the quality of shadow propagation.

In this example, the authors decided that extracting the shift count D (by calculating the number of trailing zero bits) from both operands of a multiplication instruction is too expensive, both in terms of run time and code size. We noticed that every case when application logic depends on the fact that the least significant bits of the result are zero is the result of a particular optimization and has a constant as the second operand. We implemented specialized shadow propagation logic for this case:

$$A = B * (C * 2^D) \implies A' = B' \ll D$$

In general case, multiplication is approximately instrumented with a bitwise OR:

$$A = B * C \implies A' = B'\ |\ C'$$

3.3.3 Relational comparison

Initially, shadow propagation for relational comparison $A = (B > C)$ was implemented with a bitwise OR, which made the result undefined unless both sides of the comparison are fully defined. It produced a false positive on the following code when `s->a` is not initialized:

```
struct S { int a : 3; int b : 5; };
bool f(S *s) { return s->b; }
```

The result of the conversion of the second field of S to boolean can be expressed as

```
*(unsigned char *)s > 7
```

LLVM does this transformation as an optimization, as it allows more compact encoding than otherwise extracting the 5 highest bits. In this example, `s` contains both fields of `struct S`, but the result of comparison does not depend on the value of the `a` field.

Given an *unsigned* integer value X and its shadow value (i.e. the mask of the undefined bits) X', the real value of X can be in the range of

$$[\ \text{VMin}(X, X'), \text{VMax}(X, X')\ ]$$

, where

$$\text{VMin}(X, X') = X\ \&(\sim X'), \text{VMax}(X, X') = X\ |\ X'.$$

The result of the comparison is defined if and only if it is not affected by the values of undefined bits, i.e. when intersection of intervals for both operands is empty.

For *signed* integers formulas for VMin and VMax become much more complex.

Then, for $A = (B < C)$ shadow value of A can be calculated as

$$A' = ((\text{VMin}(B, B') < \text{VMax}(C, C'))\ \text{xor}\ (\text{VMax}(B, B') < \text{VMin}(C, C')),$$

i.e. the value of A is defined iff the intervals for B and C do not intersect. Note that A' as well as A is a single-bit value.

This formula defines the exact condition when the result of the comparison depends on undefined bits. Unfortunately, it is very computationally expensive. Benchmarks show slowdown of up to 50% when all relational comparisons are instrumented this way, compared to simple bitwise OR propagation.

The authors ended up with the same compromise as in the multiplication case above. Expensive and correct propagation is used only when one of the compared values is a compile-time constant. This covers all cases when a compiler generates such comparison as an optimization.

3.3.4 Equality comparison

Similar issues exist with equality comparison instructions. Compiler-generated code sometimes uses the fact that the result of equality comparison result is defined even when part of the operands bits are not. It's important to note that the result of the comparison is defined in the following two cases (and undefined otherwise):

- B and C are fully defined (i.e. $B' = C' = 0$).
- There exists a bit position i such that $B'_i = C'_i = 0$ and $B_i \neq C_i$.

MemorySanitizer transforms $A = (B == C)$ in the following way:

$$D = B \operatorname{xor} C, \quad A = (D == 0).$$

Then, shadow of the result can be calculated as:

$$D' = B' \mid C',$$

$$A' = (!(D \,\&\, \sim D')) \,\&\&\, (D' \neq 0).$$

3.3.5 Ternary operator

Another interesting case is the instrumentation of the `select` instruction, which models C ternary operator `?:`. An important fact to note is that it is possible for the result of `select` to be defined even if the condition itself is not defined. Namely, if in $A = select\ B, C, D$ there is a bit position i such that C_i and D_i are equal and both defined, then A_i does not depend on the value of B and is always defined. This leads to the following exact shadow propagation logic:

$$A = B \,?\, C : D$$

$$A' = B' \,?\, [(C \operatorname{xor} D) \mid C' \mid D'] : [B \,?\, C' : D']$$

3.3.6 Vector instructions

Vectors are first-class types in LLVM IR, which makes most of the shadow propagation logic described above applicable verbatim. For example, parallel addition is modelled as parallel bitwise OR, and even parallel select is modelled by the exact same formular and scalar select, but operating on vector values.

Three special vector operations `extractelement`, `insertelement` and `shufflevector` that are instrumented in a straightforward way by applying the same operation to the shadow values.

3.3.7 Thread safety

In multithreaded environment, shadow update must be done concurrently with the corresponding application memory store. This is not an issue for plain store instructions because LLVM IR follows the C++ memory model in disallowing data races. If there is a happens-before relation between a store and a load from one memory location, the same relation exists between the corresponding shadow store and shadow load.

Handling of atomic operations, however, requires a different approach. Ideally, every atomic store should update the corresponding shadow memory location in an atomic way. This could be implemented by doing all atomic operation with a particular location, and corresponding shadow operations, under a common lock. Contention could be reduced using a global hash table of locks.

This approach would significantly slow down atomic operations, as in addition to the original store or load the tool would have to find, acquire and later release a lock.

MemorySanitizer implements a different, faster, approach that is designed to avoid false positive reports at the cost of possible false negative errors.

- Atomic loads are instrumented with a shadow load that follows the original instruction.
- Atomic stores are instrumented with a shadow store of *zero* value that precedes the original instruction.
- Atomic loads get acquire ordering, atomic stores get release ordering.

This way, any atomically accessed location in the program may only change from uninitialized to fully initialized state, but not the other way around.

If a store-load pair constitutes a happens-before arc, shadow store and load are correctly ordered such that the load will observe either the value that was stored, or some later value (which is always initialized).

Compare-And-Swap and Read-Modify-Write operations are instrumented in a similar way by storing a zero shadow value before the original instruction, considering the previous value to be always initialized.

3.3.8 Function calls

MemorySanitizer uses a special thread-local array to pass shadow values for function parameters from the caller to the callee.

Handling of variable argument list functions in MemorySanitizer is quite complicated due to the asymmetric low-

ering of such functions in LLVM IR. On the caller side, variable argument list functions look the same as normal functions. On the callee side, however, the details of the platform-dependent `va_list` format are exposed in the IR that is generated by the compiler frontend to access argument values. At the same time the code that sets up `va_list` is hidden behind an opaque `va_start` intrinsic call.

MemorySanitizer instruments `va_start` call to update the shadow for the resulting `va_list` structure. This instrumentation is platform-dependent and mirrors the `va_start` lowering code in the platform backend.

3.4 Run-time library

MemorySanitizer run-time library shares much common code with AddressSanitizer and ThreadSanitizer libraries. At startup it makes the lower protected area inaccessible, and maps *Shadow* and, optionally, *Origin* areas. MemorySanitizer is currently limited to Linux / x86_64, and these memory ranges (as specified in Figure 1) are always available at startup, provided that the application is linked as PIE (position-independent executable), and address space layout randomization (ASLR) is enabled.

MemorySanitizer uses the same allocator as the other Sanitizer tools. It does not add redzones around memory allocations, and does not implement memory quarantine. Allocated regions (with the exception of `calloc` regions) are marked as uninitialized, or 'poisoned'. Deallocated regions are marked uninitialized as well.

To update shadow state for memory operations done in libc library, MemorySanitizer intercepts a large subset (close to 300) of standard libc functions.

3.5 Origin tracking

UUM reports are notoriously hard to debug. By the nature of the bug, they tell that something *has not happened* (i.e. memory was not initialized), and understanding where it should have happened requires knowing the programmer's intent, which is not something an automated tool can do.

Instead, MemorySanitizer implements *origin tracking*, which helps users to understand the errors. It is very similar to a technique by the same name used in Memcheck, which is partially based on paper [7].

In origin tracking mode, MemorySanitizer associates a 32-bit *origin* value with each application value. This value serves as an identifier of a memory allocation (either heap or stack) that created the uninitialized bits this value depends on. Origin has no meaning for fully defined values.

Code instrumentation is changed to propagate origin values. For example, for a 3-argument operation $A = \mathrm{op}\ B, C, D$ origin value for A is calculated as

$$A'' = (D')\ ?\ (D'') : (C'\ ?\ C'' : B'').$$

This way A gets the origin ID of one of the undefined operands. If all operands are defined, then A is defined as well, and it's origin ID does not matter.

A special case is the ternary operator, which is represented as `select` instruction in LLVM IR. Origin ID for the ternary operator $A = B\ ?\ C : D$ is calculated as

$$A'' = (B')\ ?\ (B'') : (B\ ?\ C'' : D'').$$

When an undefined value is stored to memory, its origin ID is stored to the secondary shadow space, marked as *Origin* on Figure 1. Every 4 aligned bytes of memory can have only one origin ID associated with them. If such 4 bytes combine undefined data coming from different memory allocations, their origin ID will correspond to the last undefined store in this range.

The run-time library generates new origin identifiers on each memory allocation and keeps a mapping between that id and a description of the allocation. A concurrent hash map is used for heap memory allocations with a stack trace (as a list of memory addresses) as a key, and the newly generated origin ID as a value. This approach allows fast lookup of origin ID by stack trace to avoid creating multiple origin ID for the same stack trace. Reverse lookup of stack trace by origin ID is needed only when a use-of-uninitialized-value report is printed, which is a relatively rare occurence.

For performance reasons, stack allocations do not get a full stack trace, and only record function and variable names.

An example of origin tracking report can be found in Appendix A.

3.6 Advanced origin tracking

With origin tracking, use-of-uninitialized value reports in addition to the current stack trace include a stack trace of the allocation where this undefined value came from. Sometimes this is not enough. A value may undergo multiple memory copies and transformations that make understanding the sequence of events between the allocation and the use challenging.

MemorySanitizer implements *advanced origin tracking* mode in which it prints stack traces of all memory stores along the path from the allocation to the use of the uninitialized value.

When an undefined value is stored to memory, instead of directly storing its origin ID to the origin shadow, MemorySanitizer creates a new origin ID that corresponds to the pair of (previous origin ID, current stack trace) and stores that new id to the secondary shadow.

This effectively turns origin ID into a descriptor of a sequence of undefined stores starting with its creation (with a heap or a stack allocation). All origin IDs with the same allocation stack form a prefix tree with the allocation origin ID as the root node.

Edges of the history tree are stored in another hash map with a pair of (previous origin ID, current stack trace id) as the key and new origin ID as the value. Hash map provides de-duplication of origin identifiers: if multiple undefined values follow the same path, they are assigned the existing origin ID that encodes the exact sequence of memory stores.

Changes to instrumentation required to implement this approach are very simple. Every time an uninitialized value A is stored to memory, the corresponding origin ID is calculated by passing A'' to a run-time function.

Extra computations involved in origin tracking only need to be performed when the value being stored is undefined, which is relatively uncommon. This explains the limited extra program slowdown of this mode compared to the basic origin tracking. See Section 4.3 for more details.

In order to record memory stores in functions like `memcpy` and `memmove`, their implementation in MemorySanitizer runtime library is changed to generate new origins ids corresponding to the copy operation and write them to the secondary shadow of the destination. It is important to do this only for uninitialized memory, as origin of initialized memory is meaningless, and we may end up creating a lot of extra history nodes.

If a program often copies initialized memory, this change may result in less secondary shadow memory being paged in, and significantly reduce RAM usage. A notable example is `433.milc` benchmark in Table 2, where advanced origin tracking mode reduces total RAM usage by 17%.

3.6.1 Memory footprint of advanced origin tracking

Some programs have, practically, unlimited growth of history tree over time. One such example is `povray` in SPEC-2006, which generates store chains of average length 76, where each store can have one of more than 20 unique stack traces. All combinations of stores are possible, which gives the upper limit of 20^{76} unique histories.

We've applied two limits on the history tree growth:

- History depth. MemorySanitizer records the store chain length in the 3 highest bits of an origin identifier. Only the first 6 stores are recorded, and the rest are ignored.
- Per-stack use count. Each store stack trace is allowed to participate in the creation of a new history node a limited number of times.

As soon as one of the limits is reached, MemorySanitizer stops generating new origin IDs and propagates an unmodified origin ID, effectively falling back to basic origin tracking mode for a given store instruction. This strategy works well in terms of performance, yet we have not observed any usability issues in practice.

4. Evaluation

In this section we compare MemorySanitizer with other tools, provide CPU and memory usage statistics, and discuss deployment challenges.

4.1 Comparison

Compiler instrumentation has certain advantages over binary instrumentation used in tools like Memcheck and Dr. Memory. Intermediate program representation used by compilers carries more information about program semantics than binary code. In some cases this allows faster and simpler instrumentation which is also less prone to false positives.

For example, Memcheck goes to great lengths to instrument operations that affect x86/ARM flags register in a way that is both fast and has no false positives. Representing flags shadow in a usual, bit-for-bit way results in prohibitively slow instrumentation, as, at least on x86, a very large subset of instructions modify multiple flags. Memcheck implements a different, less precise representation for the flags register which results in false positives in some rare cases.[3]

In comparison, LLVM IR has a notion of boolean-typed temporary, `i1`, and several comparison instructions that produce such temporaries. MemorySanitizer represents `i1` shadow as `i1`, a single-bit value, which exposes extra optimization opportunities by side-stepping the bottleneck of a central flags register.

The Memcheck paper [18] states that there are "a few" false positives when compiling with optimization. Authors' experience with Chromium shows that continuous testing of a large project has very low tolerance for false positives. Stable results could be achieved only at -O1 optimization level with some extra optimizations disabled[4]. This further penalises Memcheck performance compared to compiler-based tools like MemorySanitizer.

One common type of false positives in binary instrumentation tools comes from applying C++ semantics to binary code. Logical OR operator (||) in C++ is short-circuited, i.e. in $A \mid\mid B$ sub-expression B is not evaluated unless A is false. If B does not include side-effects, it can be evaluated speculatively, and compilers would often do this to exploit instruction-level parallelism. This may result in an "impossible" report of B being used uninitialized while A is true.

One more advantage of compiler-based instrumentation is that the tool has all the names of local variables on stack. MemorySanitizer in track origins mode can tell the name of the exact stack variable that has not been initialized, while Valgrind can only point to the function name.

4.2 MemorySanitizer deployment

Deploying MemorySanitizer may be far from trivial because it requires to instrument **all** memory accesses in the program, including those that happen in pre-built binaries or in hand-written assembly.

We have set up a continuous testing process that builds LLVM using MemorySanitizer, runs all tests, and builds it again using the instrumented compiler binary (bootstrap). This process has found over 15 regressions in 10 months; in most cases the LLVM developers were notified about the bug within a few hours after introducing it. We have also

[3] https://bugs.kde.org/show_bug.cgi?id=270709

[4] For testing with Memcheck we build Chromium with the following compiler flags: -O1 -g -fno-inline -fno-omit-frame-pointer -fno-builtin -fno-optimize-sibling-calls

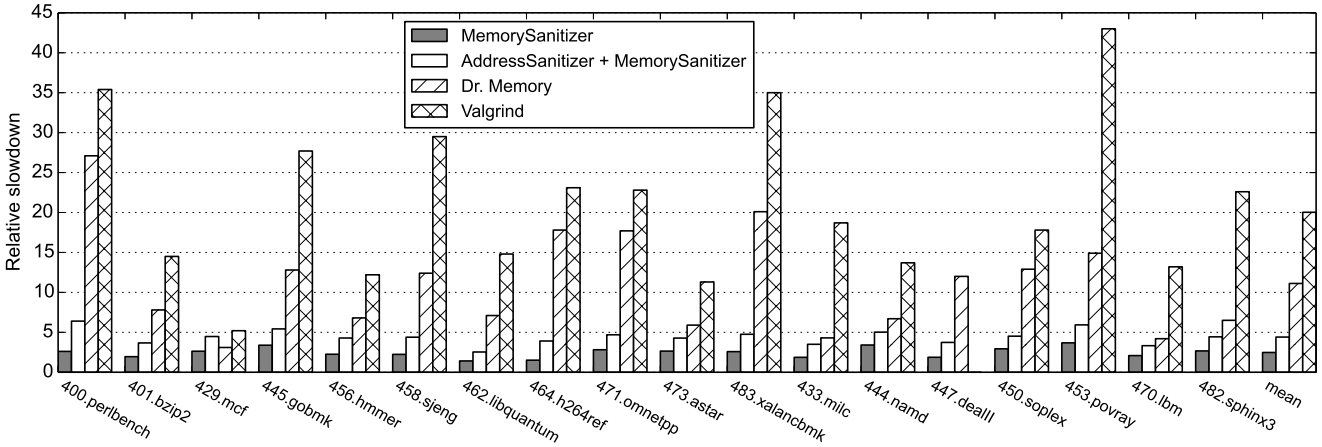

Figure 2. Performance comparison with state-of-the-art tools.

successfully built another opensource compiler, GCC, and reported one UUM bug. Making MemorySanitizer work on LLVM and GCC was trivial as both projects have neither external dependencies nor hand-written assembly.

Second, we have deployed MemorySanitizer for testing the Google server-side applications. This is a huge code base with over 100 MLOC in C++. Most of this code has neither binary-only dependencies nor hand-written assembly, so we were able to instrument the entire code for the majority of tests and detect over 500 bugs during the initial deployment. Current limitation of MemorySanitizer does not allow us to use it for tests with mixed code (e.g. Python or Java mixed with C++).

Finally, we've enabled MemorySanitizer for the Chromium browser [1] by rebuilding over 50 Linux system libraries that Chromium depends on with MemorySanitizer. In MemorySanitizer build, JavaScript is compiled into AArch64 machine instructions that are executed in the built-in AArch64 simulator, which is instrumented with MemorySanitizer — this allows the tool to observe all memory accesses coming from JavaScript. Hundreds of bugs were detected with the help of fuzzing testing on ClusterFuzz [2]. Due to the higher tool speed we've been able to deploy testing at a much larger scale than it was possible before.

4.3 Performance and memory usage

Figure 2 shows the slowdowns of different UUM detectors on SPEC 2006 [19]. We have compared the performance of MemorySanitizer with Dr. Memory and Memcheck (Valgrind). We've also included into the comparison the sum of AddressSanitizer and MemorySanitizer slowdowns, which is a combination of sanitizer tools that detects both UUM and addressability bugs, and, as such, is directly comparable to the competing tools.

MemorySanitizer measurements were done on 64-bit Linux on a 6-core Intel Xeon W3690 @ 3.47GHz machine with 24Gb RAM. The slowdown with Intel Inspector XE 2013 (flags `-collect=mi3 -knob analyze-stack=true`) on SPEC ranges from 13x to 500x (180x on average), so we do not discuss it any further. Memcheck and Dr. Memory data was taken from [9]. DealII benchmark seems to run forever under Memcheck and is excluded from Figure 2.

Table 2. Memory usage with MemorySanitizer (MB)

Benchmark	Base	MSan	M/O	M/AO
400.perlbench	661	2.05x	3.04x	3.04x
401.bzip2	849	2.00x	3.00x	3.00x
429.mcf	1676	2.00x	2.99x	2.99x
433.milc	679	1.99x	2.34x	2.00x
444.namd	46	2.28x	3.30x	3.31x
445.gobmk	28	2.07x	3.03x	3.05x
447.dealII	792	2.36x	3.40x	3.41x
450.soplex	425	2.86x	4.40x	4.40x
453.povray	4	5.22x	7.23x	8.04x
456.hmmer	25	2.76x	4.11x	4.11x
458.sjeng	175	2.01x	3.01x	3.00x
462.libquantum	96	2.03x	2.36x	2.36x
464.h264ref	64	1.88x	2.39x	2.38x
470.lbm	409	2.00x	3.00x	3.00x
471.omnetpp	169	2.15x	3.09x	3.10x
473.astar	325	2.37x	3.58x	3.58x
482.sphinx3	42	1.71x	1.85x	1.84x
483.xalancbmk	419	2.50x	3.74x	3.76x

Table 2 (**MSan** for default mode, **M/O** for origin tracking mode and **M/AO** for advanced origin tracking mode) summarises the increase in the memory usage, collected by running SPEC2006 benchmarks with `time -f %M` which prints the maximum resident set size of a process. For benchmarks that run multiple processes, we've picked the one with the highest memory usage.

Figure 3 compares the slowdown of different MemorySanitizer modes. The performance and memory usage dif-

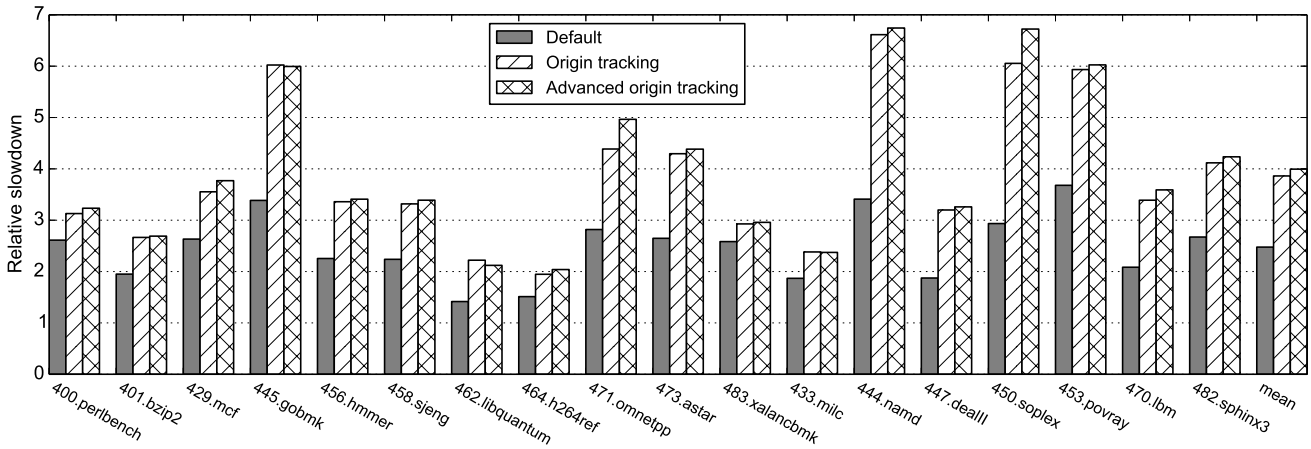

Figure 3. Performance comparison of origin tracking modes.

Table 3. Application startup time (ms).

Benchmark	Base	MSan	MSan/O	Valgrind	Valgrind/O	Dr. Memory
Clang	17	106	118	4525	6053	828
Chromium	586	898	1257	97996	158230	n/a

ference between basic and advanced origin tracking is small enough that we are considering making the latter the default origin tracking mode.

Table 3 summarises application startup times for MemorySanitizer, Valgrind, and Dr. Memory[5] using two benchmarks: compilation of a simple "hello world" C++ source file with the Clang compiler, and startup of Chromium browser measured in an HTML onload event.

As you can see, tools based on run-time binary instrumentation pay extremely large startup cost on large binaries, while MemorySanitizer does not. This feature is important when a tool is used to test large binaries with short-running tests, such as on ClusterFuzz [2].

5. Future work

Improving performance and memory consumption of dynamic testing tools, including MemorySanitizer, is an endless task. Comprehensive static analysis could eliminate some of the run-time checks, and researchers already try that for MemorySanitizer [20]. Memory consumption in the default mode can hardly be decreased, but in the origin tracking modes it is still possible.

MemorySanitizer can be extended to report more kinds of bugs. For example, a bug where a C++ object is used after its destructor is called but before the memory is deallocated.

To avoid false positives when parts of code are not instrumented, it is possible to combine compiler instrumentation with static or dynamic binary instrumentation. At least one such attempt has been made [12], but the resulting tool was too complex and slow compared to pure MemorySanitizer.

Last but not least, the software developer community needs to embrace the idea of having *all* of the source code available for (re)compilation. This is required not just for MemorySanitizer: other static and dynamic analysis tools and optimizing compilers will benefit from it. For example, usage of Intel MPX [11] instruction set extension could be greatly simplified if the entire program is being instrumented. With a very few exceptions the software libraries on Linux can be recompiled.

Conclusions

This paper presents MemorySanitizer, a tool that detects uses of uninitialized memory (UUMs). Traditionally, UUM detectors use binary instrumentation, but MemorySanitizer uses compiler instrumentation that allows it to be an order of magnitude faster, have no startup penalty, and provide more detailed reports compared to other state of the art tools. MemorySanitizer has an optional mode that tracks origins and copies of the uninitialized values to provide easier to understand error messages. We have deployed MemorySanitizer in several large scale software projects and demonstrated the tool's advantages.

MemorySanitizer is open-source and a part of LLVM [3] compiler suit.

[5] Dr. Memory does not officially support Linux/x64; we were able to run it on clang, but not on Chromium.

References

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

[2] ClusterFuzz. `http://blog.chromium.org/2012/04/fuzzing-for-security.html`.

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

[4] ThreadSanitizer. `https://code.google.com/p/thread-sanitizer/`.

[5] WG21 N3914. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3914.html`.

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

[7] Michael D. Bond, Samuel Z. Guyer, Nicholas Nethercote, Stephen W. Kent, and Kathryn S. Mckinley. Tracking bad apples: reporting the origin of null and undefined value errors. In *In Proc. of the ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications*, pages 405–422, 2007.

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

[9] 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.

[10] Intel. Intel Inspector XE. `http://software.intel.com/en-us/intel-inspector-xe`.

[11] Intel. Intel MPX. `https://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions`.

[12] Timur Iskhodzhanov, Reid Kleckner, and Evgeniy Stepanov. Combining compile-time and run-time instrumentation for testing tools. In *Programmnye produkty i sistemy*, volume 3, pages 224–231, 2013.

[13] ISO. *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. International Organization for Standardization, Geneva, Switzerland, February 2012.

[14] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM.

[15] 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.

[16] 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.

[17] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 28–28, Berkeley, CA, USA, 2012. USENIX Association.

[18] 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.

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

[20] Ding Ye, Yulei Sui, and Jingling Xue. Accelerating dynamic detection of uses of undefined values with static value-flow analysis. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 154:154–154:164, New York, NY, USA, 2014. ACM.

A. MemorySanitizer report example

```
1  int arr[2];
2  void shift() { arr[1] = arr[0]; }
3  void push(int *p) {
4    shift();
5    arr[0] = *p;
6  }
7  int pop() {
8    int x = arr[1];
9    shift();
10   return x;
11 }
12 void func1() {
13   int local_var;
14   push(&local_var);
15 }
16 int main() {
17   func1();
18   shift();
19   return pop();
20 }
```

The following report was obtained by compiling the above source with the latest Clang compiler (SVN r214393) with `-fsanitize-memory-track-origins=2 -g -fsanitize=memory` command line flags.

```
WARNING: MemorySanitizer: use-of-uninitialized-value
    #0 0x7fa14df37e1d in main test.c:19:10
  Uninitialized value was stored to memory at
    #0 0x7fa14df37a57 in pop() test.c:8:3
    #1 0x7fa14df37dd5 in main test.c:19:10
  Uninitialized value was stored to memory at
    #0 0x7fa14df37733 in shift() test.c:2:16
    #1 0x7fa14df37dbb in main test.c:18:3
  Uninitialized value was stored to memory at
    #0 0x7fa14df3793f in push(int*) test.c:5:3
    #1 0x7fa14df37b2f in func1() test.c:14:3
    #2 0x7fa14df37db6 in main test.c:17:3
  Uninitialized value was created by an allocation of
  'local_var' in the stack frame of function 'func1'
    #0 0x7fa14df37ad0 in func1() test.c:12
```