Memory Tagging and how it improves C/C++ memory safety

Kostya Serebryany, Google

April 2018

https://arxiv.org/pdf/1802.09517.pdf

Agenda

- C++ memory safety bugs, AddressSanitizer (ASAN)
- Memory tagging concept
- Implementation: LLVM HWASAN, SPARC ADI
- Memory tagging as security mitigation

Memory Safety in C++

- Heap-use-after-free
- Heap-buffer-overflow
- Stack-buffer-overflow
- Stack-use-after-return
- Stack-use-after-scope
- Global-buffer-overflow
- Use-of-uninitialized-memory
- ~~Intra-object-buffer-overflow~~

Heap-use-after-free, Heap-buffer-overflow

```
int *p = new char[20];

p[20] = …  // OMG

delete [] p;

p[0] = …   // OMG
```

AddressSanitizer (ASAN)

- Shadow memory: every 8 bytes are mapped to 1 byte metadata
- Compiler instrumentation checks the metadata on access
- Relies on **redzones** to catch heap-buffer-overflow
- Relies on **quarantine** (delayed reuse) to catch use-after-free
- Valgrind/Memcheck: similar concept, different trade offs

ASAN (redzones, quarantine)

```
int *p = new char[20];
```

[-8:-1]	[0:7]	[8:15]	[16:23]	[24:30]

```
p[20] = ...   // OMG

delete [] p;
```

[-8:-1]	[0:7]	[8:15]	[16:23]	[24:30]

```
p[0] = ...       // OMG
```

ASAN’s Problems

- **~2x Memory overhead**
  - Shadow
  - Redzones
  - Quarantine

- Buffer overflows:
  - may jump over redzone

- Use-after-free
  - may “outlive” quarantine

Memory Tagging (MT) in one slide

- 64-bit architectures only
- Every aligned TG bytes have a TS-bit tag
    - TG = tagging granulariy, TS = tag size
    - E.g. every 16 bytes of memory have a 8-bit tag (TG=16, TS=8)
- Every pointer has a 8-bit tag in the top byte
- Memory allocation tags memory & pointers with the same tag
- Loads/stores fail on tag mismatch
- Detects use-after-free and buffer-overflow (heap, stack, globals)

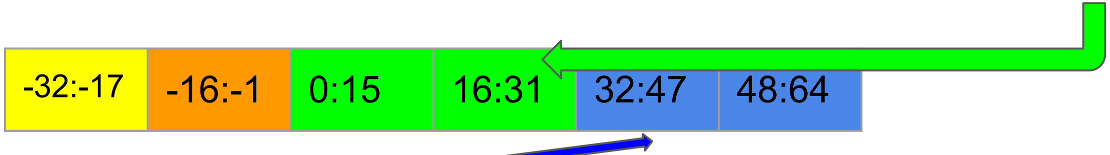
Heap-use-after-free, Heap-buffer-overflow

- Malloc:
  - Align to TG
  - Choose a tag
  - **Tag the memory**
  - Tag the pointer

- Free (optional):
  - Re-tag the memory

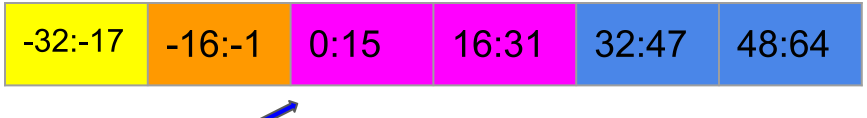
Memory Tagging (TG=16, TS=8)

```
int *p = new char[20]; // 0xab007fffffff1240
```




```
p[32] = ... // OMG

delete [] p;
```




```
p[0] = ...  // OMG
```

Probability of bug detection, general case

- $(2^{TS}-1)/(2^{TS})$
- TS = 8: 255/256 = 99.6%
- TS = 4: 15/16 = 93.7%

Precision of buffer overflow detection

```
int *p = new char[20];

p[20] = … // undetected (same granule)

p[32] = … // detected (*)

p[-1] = … // detected (*)

p[100500] = … // detected with high probability
```

Tag assignment strategies

- Random
- Dedicated “match-none” tag:
  - 100% off-by-one (linear) buffer overflow detection, requires redzones
  - 100% use-after-free-before-realloc detection
- Odd tags for odd chunks (and even tags for even chunks)
  - 100% off-by-one (linear) buffer overflow detection
  - Reduces the number of tag bits useful for use-after-free

False positives

Don’t happen

stack-{buffer-overflow,use-after-return,use-after-scope}

- Compiler instrumentation to tag/untag local variables
- Same otherwise

global-buffer-overflow

- Tag globals and their addresses

Using the Top Byte of a Pointer

- On x86_64:
  - very hard, need to instrument all memory accesses

- On AArch64:
  - easy, thanks to [top-byte-ignore]()

- Other uses of top-byte-ignore in existing software?
  - Android, Chrome: OK
  - Swift and Objective-C: uses do not overlap with C++ pointers (??)

MT vs ASAN

- MT:
  - Small RAM overhead
    - 6% with TG=16 TS=8
    - 0.7% with TG=64 TS=4
  - Detection of buffer overflows far from bounds
  - Detection of use-after-free long after deallocation

- ASAN:
  - Precise 1-byte buffer-overflow detection
  - More portable (32-bit, non-aarch64)

HWASAN (HardWare-assisted ASAN, Clang/LLVM)

- AArch64: real thing
- x86_64: toy, needs to instrument all loads/stores
- TG=16, TS=8; 2x CPU, **6% RAM,** ~2.5x code size

```
// int foo(int *a) { return *a; }
// clang -O2 --target=aarch64-linux -fsanitize=hwaddress -c load.c
    0:      08 dc 44 d3     ubfx    x8, x0, #4, #52   // shadow address
    4:      08 01 40 39     ldrb    w8, [x8]          // load shadow
    8:      09 fc 78 d3     lsr     x9, x0, #56       // address tag
    c:      3f 01 08 6b     cmp     w9, w8            // compare tags
   10:      61 00 00 54     b.ne    #12               // jump on mismatch
   14:      00 00 40 b9     ldr     w0, [x0]          // original load
   18:      c0 03 5f d6     ret
   1c:      40 20 21 d4     brk     #0x902            // trap
```

Kernel-HWASAN (Linux)

Same thing, patches under review

SPARC ADI

- Available in SPARC M7/M8 CPUs since ~2016
- TG=64, TS=4
- Tl;Dr:
  - works great
  - low overhead
  - heap bugs only (no stack-buffer-overflows)

ADI: precise vs imprecise

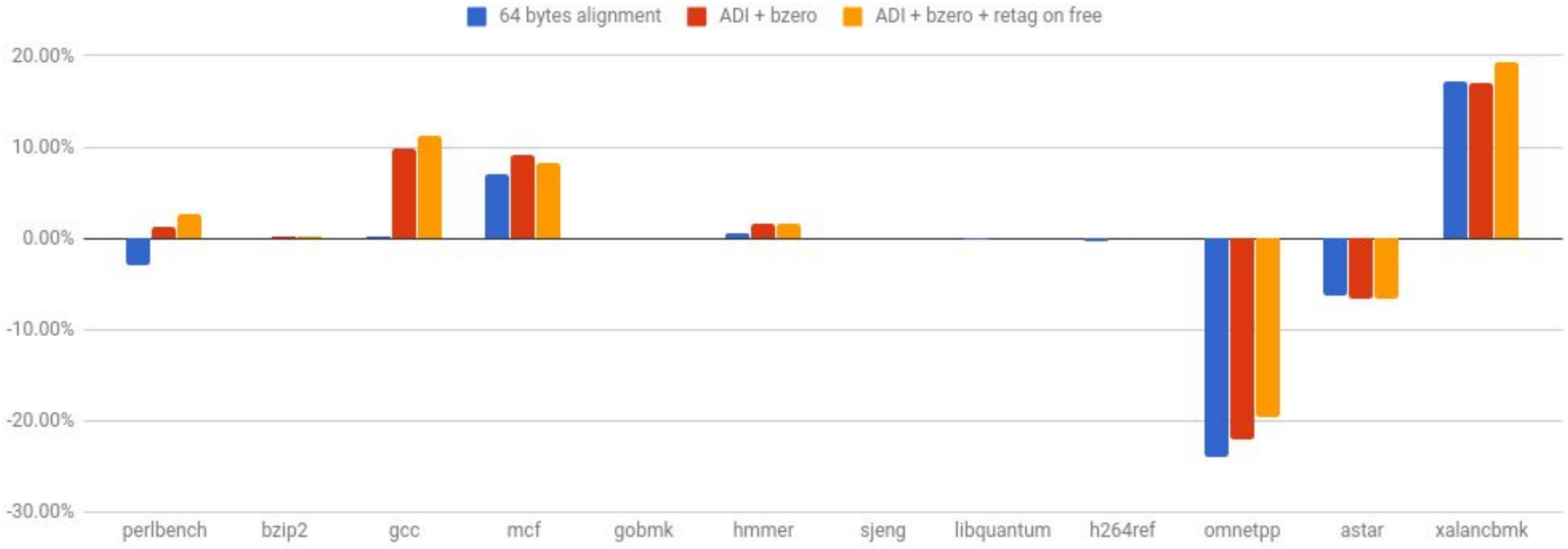
- Precise mode:
  - Tag mismatch on store causes immediate trap
  - Expensive, great for debugging

- Imprecise mode
  - Tag mismatch on store causes a trap some time later
  - Very low overhead

- Loads are always precise

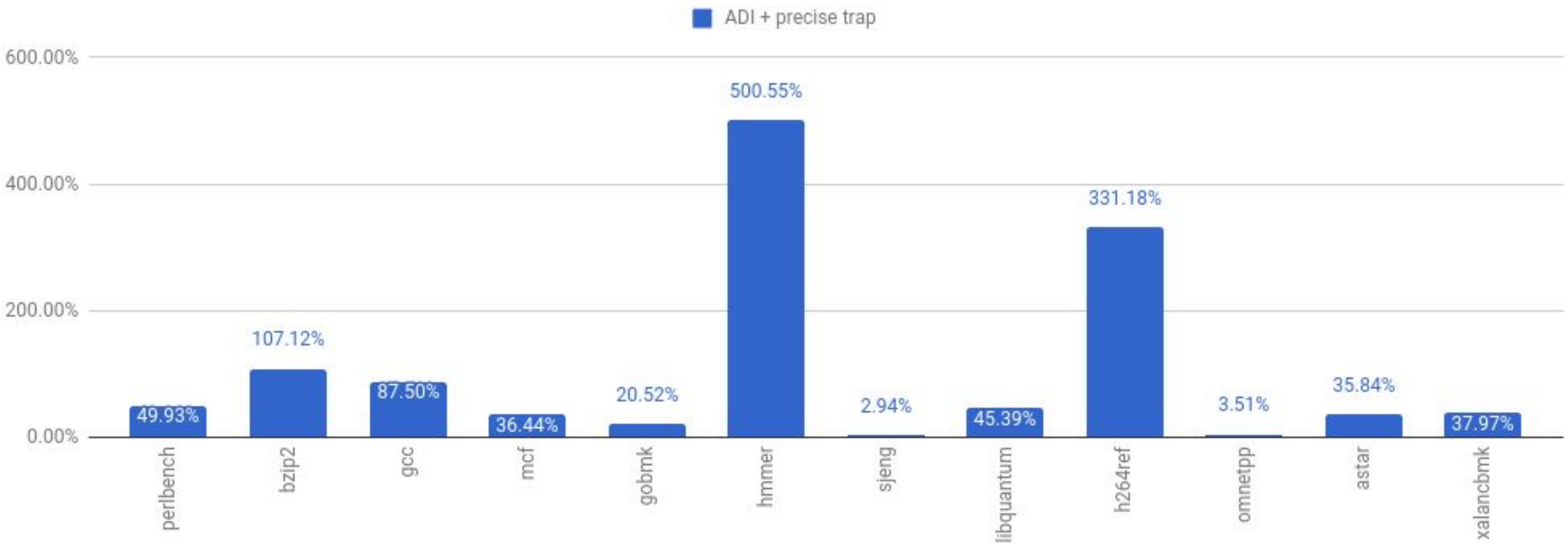
ADI overhead (imprecise)

Overhead: 64-byte alignment and (less) tagging memory on malloc



ADI overhead (precise)

Stores become very expensive



Initializing memory

Tagging heap memory has the same cost as Tagging and Initializing

MT is good for

- Testing
  - Alternative to ASAN, consumes much less RAM
- Bug detection in production
  - Crowd-sourced bug detection
  - If CPU, RAM, Code size overheads are tolerable
  - SPARC ADI - yes, HWASAN - hm, maybe
- Security mitigation: not clear, probably.

Mitigation: linear-buffer-overflow

- Linear-buffer-overflow: a granule adjacent to the buffer is accessed
  - Heartbleed, Ghostbug, Dnsmasq, Total Meltdown, CVE-2018-5146 in Firefox, Venom

- Allocator ensures that adjacent allocations have different tags

- Exploits are reliably prevented

Mitigation: use-after-free, non-linear buffer overflows

- An attack succeeds with low probability
  - 7% with TS=4
  - 0.4% with TS=8

- Will discourage most (some?) attackers:
  - Unreliable exploit
  - User and vendor get notified on failed attempts
  - Vendor gets actionable and *bucketizable* bug report

Reliably bypassing MT

- Leak and/or overwrite address tags
  - memory tags too?
- Traditional leaks/overwrites are via memory corruption or uninit
  - But they are protected from, catch 22
- Other classes of bugs need to be mitigated separately
  - Artitmetic overflows (integeres and pointers)
  - Intra-object-buffer-overflows
  - Type confusion
  - Logical bugs
  - Side channel

Reliably bypassing MT requires to have access to unmitigated bugs of other class(es).

Mitigation: uninitialized memory

No more uninitialized memory,
but we can also do it today

Home work

Ask your favourite CPU vendor to implement memory tagging

Analyze you favourite exploit: is it preventable by MT?

Q&A

https://arxiv.org/pdf/1802.09517.pdf

https://clang.llvm.org/docs/HardwareAssistedAddressSanitizerDesign.html