

Memory Tagging:

how it improves C/C++ memory safety.

Compiler perspective.

Kostya Serebryany, Evgenii Stepanov, Vlad Tsyrklevich

(Google)

Oct 2018

Agenda

- ARM v8.5 Memory Tagging Extension
- Related compiler/optimizer challenges

C & C++ memory safety is a mess

- Use-after-free / buffer-overflow / uninitialized memory
- > 50% of High/Critical security bugs in Chrome & Android
- Not only security vulnerabilities
 - crashes, data corruption, developer productivity
- AddressSanitizer (ASAN) is not enough
 - Hard to use in production
 - Not a security mitigation

ARM Memory Tagging Extension (MTE)

- [Announced](#) by ARM on 2018-09-17
- Doesn't exist in hardware yet
 - Will take several years to appear
- “Hardware-ASAN on steroids”
 - RAM overhead: 3%-5%
 - CPU overhead: (*hoping for*) low-single-digit %

ARM Memory Tagging Extension (MTE)

- 64-bit only
- Two types of tags
 - Every aligned 16 bytes of memory have a 4-bit tag stored separately
 - Every pointer has a 4-bit tag stored in the top byte
- LD/ST instructions check both tags, raise exception on mismatch
- New instructions to manipulate the tags

Allocation: tag the memory & the pointer

- Stack and heap
- Allocation:
 - Align allocations by 16
 - Choose a 4-bit tag (random is ok)
 - Tag the pointer
 - Tag the memory (optionally initialize it at no extra cost)
- Deallocation:
 - Re-tag the memory with a different tag

Heap-buffer-overflow

```
char *p = new char[20]; // 0xa007ffffff1240
```



Heap-buffer-overflow

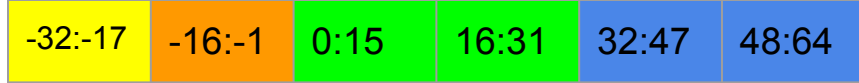
```
char *p = new char[20]; // 0xa007ffffff1240
```



```
p[32] = ... // heap-buffer-overflow █ ≠ █
```

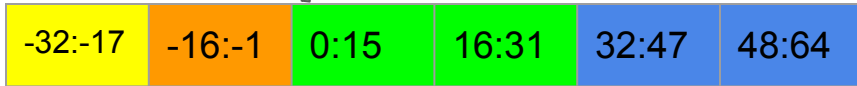

Heap-use-after-free

```
char *p = new char[20]; // 0xa007ffffff1240
```



Heap-use-after-free

```
char *p = new char[20]; // 0xa007ffffff1240
```



```
delete [] p; // Memory is retagged green ⇒ magenta
```



```
p[0] = ... // heap-use-after-free green ≠ magenta
```

Probabilities of bug detection

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

```
p[20] // undetected (same granule)
```

```
p[32], p[-1] // 93%-100% (15/16 or 1)
```

```
p[100500] // 93% (15/16)
```

```
delete [] p; p[0] // 93% (15/16)
```

BTW: other existing implementations

- SPARC ADI
 - Exists in real hardware since ~2016 (SPARC M7/M8 CPUs)
 - 4-bit tags per 64-bytes of memory
 - Great, but high RAM overhead due to 64-byte alignment
- LLVM HWASAN
 - Software implementation similar to ASAN (LLVM ToT)
 - 8-bit tags per 16-bytes of memory
 - AArch64-only (uses [top-byte-ignore](#))
 - Overhead: **6% RAM**, 2x CPU, 2x code size

New MTE instructions ([docs](#), [LLVM patch](#))

IRG Xd, Xn

Copy Xn into Xd, insert a random 4-bit tag into Xd




bit manipulations with the address tag

ADDG Xd, Xn, #<immA>, #<immB>

Xd := Xn + #immA, with address tag modified by #immB.

STG [Xn], #<imm>

Set the memory tag of [Xn] to the tag(Xn)



storing the memory tag

STGP Xa, Xb, [Xn], #<imm>

Store 16 bytes from Xa/Xb to [Xn] and set the memory tag of [Xn] to the tag(Xn)

Relax and wait for the hardware?



No, compiler writers need to reduce the overhead



MTE overhead

- Extra logic inside LD/ST (fetching the memory tag)
 - Software can't do much to improve it (???)
- Tagging heap objects
 - CPU: malloc/free become $O(\text{size})$ operations
- Tagging stack objects (optional, but desirable)
 - CPU: function prologue becomes $O(\text{frame size})$
 - Stack size: local variables aligned by 16
 - Code size: extra instructions per function entry/exit
 - Register pressure: local variables have unique tags, not as simple as [SP, #offset]

Compiler-optimizations for MTE

Malloc zero-fill (1)

```
struct S { int64_t a, b; };  
S *foo() { return new S{0, 0}; }
```

b1 _Znwm

stp xzr, xzr, [x0]

b1 _Znwm

Malloc zero-fill (2)

```
struct S { int64_t a, b; };  
S *foo() { return new S{1, 2}; }
```

```
b1 _Znwm  
mov x3, 1 // (*)  
mov x2, 2  
stp x3, x2, [x0]
```

```
b1 _Znwm_no_tag_memory  
mov x3, 1  
mov x2, 2  
stgp x3, x2, [x0]
```

(*) Generated by GCC. LLVM produces worse code. [BUG 39170](#)

Malloc to stack conversion (see [Hal's talk](#))

- By itself makes things worse
 - Still need to tag memory, but adds code bloat
- Beneficial if tagging can be completely avoided
 - (heap-to-stack-to-registers)
- Could be combined with stack safety analysis (???)

Simple stack instrumentation

```
void foo() {  
    int a;  
    bar(&a);  
}  
...  
sub sp, sp, #16  
irg x0, sp    // Copy sp to x0 and insert a random tag  
stg [x0]      // Tag memory with x0's tag  
bl bar  
stg [sp], #16 // Before exit, restore the default  
...
```

Rematerializable stack pointers

```
void foo() {  
    int a, b, c; ...  
    bar(&a); bar(&b); bar(&c);  
}
```

```
irg x19, sp // “base” pointer with random tag  
...  
addg x0, x19, #16, #1 // address-of-a with semi-random tag  
bl bar  
addg x0, x19, #32, #2 // address-of-b with semi-random tag  
bl bar
```

Store-and-tag

```
void foo() {  
    int a = 42;  
    bar(&a);  
}
```

```
irg x0, sp  
mov w8, #42  
stgp x8, xzr, [x0] // store pair and tag memory  
bl bar
```

Unchecked loads and stores

```
int foo() {  
    int a;  
    bar(&a);  
    return a;  
}
```

```
irg x0, sp  
stg [x0]  
bl bar          // clobbers X0, but that's OK ...  
ldr w0, [sp]    // SP-based LD/ST do not check tags! (#imm offset)
```


Static stack safety analysis

- Do we need to tag an address-taken local variable?
 - Is buffer overflow possible?
 - Is use-after-return possible?
 - (Optional): is use of uninitialized value possible?
- Intra-procedural analysis is unlikely to help much
- Inter-procedural analysis:
 - Context-insensitive offset range and escape analysis for pointers in function arguments.
 - ~25% local variables (by count) proven safe; up to 60% with (Thin)LTO.
 - Patches are coming! (first one: <https://reviews.llvm.org/D53336>)

Challenge: how to test the stack safety analysis?

- Unittests for sure, but never enough
- We remove the checks that fire extremely rare, no good test suite
 - Similar problem is e.g. for bounds check removal in Java
- Use analysis in ASAN but do not eliminate the checks: report bugs in a special way and notify developers (us)

More optimizations for MTE?

- Will these optimizations be useful for something else?
- What other optimizations are possible?
- Can we reuse/repurpose any existing optimizations?

More uses for MTE?

- Infinite Watchpoints?
- Race Detection (like in [DataCollider](#))?
- Type Confusion Sanitizer? (for non-polymorphic types)
- Garbage Collection?
- ???

Summary

- ARM MTE makes C++ memory-safer
- Small, but non-zero overhead
- Compilers must reduce the overhead
- ALSO: Please ask your CPU vendor to implement MTE