Top-byte-ignore & memory tagging: does RISC-V care?

Kostya Serebryany kcc@google.com
2019-11

Agenda

- C/C++ Memory Safety is a mess
  - Does RISC-V ecosystem care? Or will care?
  - ASAN
- AArch64 top-byte-ignore (TBI)
  - HWASAN
- AArch64 v8.5 Memory Tagging Extension (MTE)
- Asks for RISC-V
  - TBI (or similar) in short term
  - MTE (or similar) in longer term

C & C++ memory safety is an industry-wide crisis

- Use-after-free / buffer-overflow / …
- > 50% of High/Critical security bugs in Chrome & Android
- Not only security vulnerabilities
  - crashes, data corruption, developer productivity
- More data: iSecCon'18

How much does RISC-V care? Or will care?

- Is RISC-V planing to intrude into mobile or laptop market in 5 years?
- Do the existing users have large C/C++ code bases?
- Is security, stability, and developer productivity a concern?
- How likely is that all code running on RISC-V will be written in safe Rust?
  - Including the OS kernels

AddressSanitizer (ASAN)

- Dynamic testing tool based on compiler instrumentation
  - In LLVM, GCC, and MSVC
  - Linux, Windows, MacOS, Android, iOS, ChromeOS, *BSD, …
  - {X86, Arm, Power, Sparc, ...} x {32-bit, 64-bit}

- But seemingly not on RISC-V
  - Any interest? Demand?

ASan report example: use-after-free

```
int main(int argc, char **argv) {
   int *array = new int[100];
   delete [] array;
   return array[argc]; } // BOOM
% clang++ -O1 -fsanitize=address a.cc && ./a.out
==30226== ERROR: AddressSanitizer heap-use-after-free
READ of size 4 at 0x7faa07fce084 thread T0
   #0 0x40433c in main a.cc:4
0x7faa07fce084 is located 4 bytes inside of 400-byte region
freed by thread T0 here:
   #0 0x4058fd in operator delete[](void*) _asan_rtl_
   #1 0x404303 in main a.cc:3
previously allocated by thread T0 here:
   #0 0x405579 in operator new[](unsigned long) _asan_rtl_
   #1 0x4042f3 in main a.cc:2
```

Challenges with using ASAN

- Separate build
- 2x-3x RAM overhead
- 2x-3x CPU overhead
- 2x-3x Code Size overhead

HWASAN - newer variant of ASAN

- Separate build
- ~~2x-3x~~ < 10% RAM overhead
- 2x-3x CPU overhead
- 2x-3x Code Size overhead

HW in *HW*ASAN is Arm's top-byte-ignore (TBI)

- Shadow memory:
  - Every 16 bytes of application memory have 1 byte metadata stored separately
- Malloc:
  - Align allocations by 16
  - Generate the Tag (e.g. a random 8-bit value)
  - Put the Tag into the top byte of the allocated address - ignored by loads & stores ⇐ TBI
  - Set the metadata of all 16-byte chunks of the allocated memory to the Tag
- Free:
  - Reset the metadata to another Tag
- Compiler:
  - Instrument all loads/stores to check "Address Tag == Memory Tag"

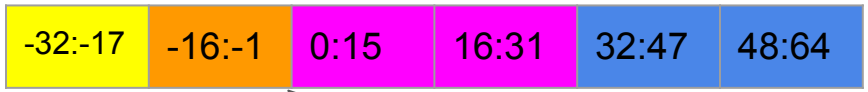
Heap-use-after-free

```
char *p = new char[20]; // 0xa007ffffffff1240
```

Heap-use-after-free

```
char *p = new char[20]; // 0xa007ffffffff1240
```


```
delete [] p;  // retagged green ⇒ purple
```




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

Heap-buffer-overflow

```
char *p = new char[20]; // 0xa007ffffffff1240
```

Heap-buffer-overflow

```
char *p = new char[20]; // 0xa007ffffffff1240
```


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

HWASAN marketing slide

- Builds and runs Android
  - Kernel
  - Userspace
  - Apps, e.g. Chrome
- Fast enough to be used as every day device (on Pixel 4)
- Enabled in pre-submit testing
- Found 120+ bugs, mostly in “dogfood”

ASK #1 (short term): something like Arm TBI

- 64-bit only
- Loads & stores ignore the top address byte
  - Variants possible, e.g. only top-4-bits or top-7-bits
- Ok to have on/off switch

Arm Memory Tagging Extension (MTE)

- Announced by Arm on 2018-09-17
  - 2019-09 blog post

- Doesn’t exist in hardware yet
  - Will take several years to appear
  - Model is already available

- Like HWASAN, but real HW this time

Arm MTE

- 4-bit tags
  - Address Tag: 4 out of 8 top bits
  - Memory Tag: stored separately, architecture does not define how
    - Most likely: tags in separate memory but joined with data in caches
- LD/ST checks that the tags match
- Synchronous and asynchronous (fast) modes

Arm MTE

- ~~Separate build~~
- ~~2x-3x~~ ~~< 10%~~ < 5% RAM overhead
- ~~2x-3x~~ *Small* CPU overhead
  - YMMV, but hoping for < 5%
- ~~2x-3x~~ Zero Code Size overhead
  - < 5% if stack is instrumented
- Can be used in production, always on or with sampling
- Also a security mitigation

Ask #2 (long term): something like Arm MTE

- 16 byte granularity is important
  - Otherwise allocation overalignment is too expensive
- Can have synchronous and asynchronous (faster) modes
- Useful even with large slowdown
  - Solves RAM & Size overhead and separate build
- Relatively cheap software bring-up as MTE paves the way
  - Android support already in progress

Pleeeeease!

TBI and MTE are Arm’s competitive advantages

See also:

- iSecCon’18
- PLEMM’19