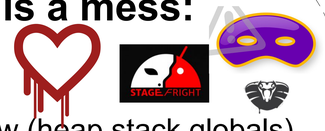
Hardware Memory Tagging makes C++ Memory-Safer

C/C++ Memory Safety is a mess:



> 20% of all C/C++ bugs and
> 50% of all vulnerabilities are
use-after-free or buffer-overflow (heap,stack,globals)

AddressSanitizer (ASAN) is not enough

- Hard to use in production:
  2x overhead in CPU, RAM, Code Size
- Weak as a mitigation:
  easy to bypass redzones & quarantine

A better scheme: Memory Tagging

- Two types of tags
  - Every aligned 16 bytes of memory have a 4-bit tag
  - Every 64-bit pointer has a 4-bit tag (in the top byte)
- LD/ST instructions SEGV on tag mismatch
- New instructions to set the tags

Existing Implementations

SPARC ADI: 4-bit tag per 64-bytes
Available in *hardware* since ~2016, SPARC M7/M8
Heap-only; 5%-20% RAM overhead due to alignment

HWASAN (Hardware ASAN): 8-bit tags per 16 bytes
Software (compiler) instrumentation, similar to ASAN
Uses AArch64-only feature “top-byte-ignore”
Limited applicability on x86_64 (needs to see all loads/stores)

ARM Memory Tagging Extension (MTE)

Announced by ARM on 2018-09-17

```
char *P = new char[20]; //0xa0007ffffffff1240
// Pointer P & memory P[0:31] are tagged with
```

-32:-17	-16:-1	0:15	16:31	32:47	48:64

```
P[32] = ... // heap-buffer-overflow ≠
delete [] P;  // Memory is retagged ⇒
```

-32:-17	-16:-1	0:15	16:31	32:47	48:64

```
P[0] = ...  // heap-use-after-free ≠
```

MTE is better than ASAN

- RAM overhead: 3%-5% (vs ~ 2x)
- CPU overhead: *expected* low-single-digit %
- Detects buffer overflows far from bounds
- Detects use-after-free long after deallocation

Usage Models:

- Regular testing/fuzzing (faster ASAN)
- Crowdsourced testing in production
- Always-on security mitigation

Homework: Ask your CPU vendor to implement Memory Tagging

Desired Compiler Optimizations

- Malloc now zero-fills and sets memory tags

```
struct S { int64_t a, b; };
// Redundant initialization after new()
S *foo() { return new S{0, 0}; }
// Redundant initialization/tagging in new()
S *bar() { return new S{1, 2}; }
```

- Malloc costs more: need malloc-to-stack
- Compact & fast stack instrumentation

`void f() {`
`  int a;`
`  bar(&a);`
`}`
	`irg x0, sp`
	`stg [x0]`
	`bl bar`
	`stg [sp], #16`
		`// x0 := sp w/ random tag`
		`// Tag the memory`
		
		`// Before exit, restore the default tag`

`int f() {`
`  int a = 42;`
`  bar(&a);`
`  return a;`
`}`
	`irg x0, sp`
	`mov w8, #42`
	`stgp x8, xzr, [x0]`
	`bl bar`
	`ldr w0, [sp]`
	`stg [sp], #16`
		
		`// Initialize & tag`
		
		`// ld/st [SP, #imm] does not check tags`

- Static inter-procedural stack safety analysis
  - Do we need to tag a local variable?
  - Is buffer overflow possible?
  - Is use-after-return possible?
  - (Optional): is use of uninitialized value possible?

Kostya Serebryany kcc@google.com kayseesee@
Evgenii Stepanov eugenis@google.com
Vlad Tsyrklevich vtsyrklevich@google.com vlad902@
See also: https://arxiv.org/pdf/1802.09517.pdf