Improving C/C++ memory safety

using sanitizers, hardware-assisted sanitizers and memory tagging

speaker: Julian Pszczołowski

C/C++ memory safety

C/C++ memory safety

- 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
- ...

```
char *p = new char[20];
p[20] = …; // BUG
delete [] p;
p[0] = …; // BUG
```

Chrome Releases July 24, 2018

All High CVEs are memory safety bugs

[$5000][850350] High CVE-2018-6153: Stack buffer overflow in Skia. *Reported by Zhen Zhou ...*
[$3000][848914] High CVE-2018-6154: Heap buffer overflow in WebGL. *Reported by Omair on 2018-06-01*
[$N/A][842265] High CVE-2018-6155: Use after free in WebRTC. *Reported by Natalie Silvanovich...*
[$N/A][841962] High CVE-2018-6156: Heap buffer overflow in WebRTC. Reported by Natalie Silvanovich ...
[$N/A][840536] High CVE-2018-6157: Type confusion in WebRTC. Reported by Natalie Silvanovich ...
[$2000][841280] Medium CVE-2018-6158: Use after free in Blink. *Reported by Zhe Jin (金哲)...*
[$2000][837275] Medium CVE-2018-6159: Same origin policy bypass in ServiceWorker.*Reported by Jun Kokatsu ...*
[$1000][839822] Medium CVE-2018-6160: URL spoof in Chrome on iOS. *Reported by evi1m0 ...*
[$1000][826552] Medium CVE-2018-6161: Same origin policy bypass in WebAudio.*Reported by Jun Kokatsu ...*
[$1000][804123] Medium CVE-2018-6162: Heap buffer overflow in WebGL. *Reported by Omair on 2018-01-21*
[$500][849398] Medium CVE-2018-6163: URL spoof in Omnibox. *Reported by Khalil Zhani on 2018-06-04*
[$500][848786] Medium CVE-2018-6164: Same origin policy bypass in ServiceWorker.*Reported by Jun Kokatsu*
[$500][847718] Medium CVE-2018-6165: URL spoof in Omnibox. *Reported by evi1m0 of Bilibili Security ...*
[$500][835554] Medium CVE-2018-6166: URL spoof in Omnibox. *Reported by Lnyas Zhang on 2018-04-21*
[$500][833143] Medium CVE-2018-6167: URL spoof in Omnibox. *Reported by Lnyas Zhang on 2018-04-15*
[$500][828265] Medium CVE-2018-6168: CORS bypass in Blink. *Reported by Gunes Acar and Danny Y. Huang of Princeton University, ...*
[$500][394518] Medium CVE-2018-6169: Permissions bypass in extension installation .*Reported by Sam P on 2014-07-16*
[$TBD][862059] Medium CVE-2018-6170: Type confusion in PDFium. *Reported by Anonymous on 2018-07-10*
[$TBD][851799] Medium CVE-2018-6171: Use after free in WebBluetooth. *Reported by amazon@mimetics.ca on 2018-06-12*
[$TBD][847242] Medium CVE-2018-6172: URL spoof in Omnibox. *Reported by Khalil Zhani on 2018-05-28*
[$TBD][836885] Medium CVE-2018-6173: URL spoof in Omnibox. *Reported by Khalil Zhani on 2018-04-25*
[$N/A][835299] Medium CVE-2018-6174: Integer overflow in SwiftShader. *Reported by Mark Brand of Google Project Zero on 2018-04-20*
[$TBD][826019] Medium CVE-2018-6175: URL spoof in Omnibox. *Reported by Khalil Zhani on 2018-03-26*
[$N/A][666824] Medium CVE-2018-6176: Local user privilege escalation in Extensions.*Reported by Jann Horn of Google Project Zero on 2016-11-18*

Every 6-8 weeks on https://chromereleases.googleblog.com, since ~ 2011

C/C++ memory safety

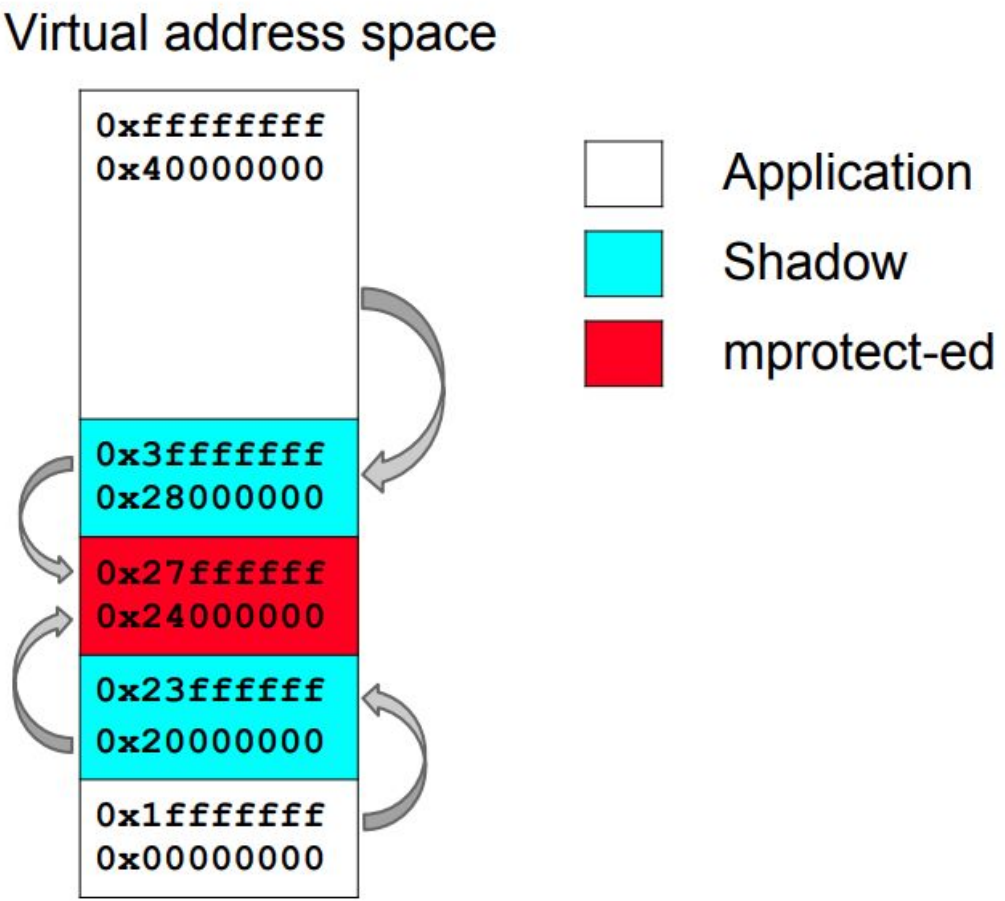
- *-use-after-free / *-buffer-overflow / uninitialized memory
  - over 50% of High/Critical security bugs in Chrome & Android
  - 70% of all security issues addressed in Microsoft products are caused by violations of memory safety
- Not only security vulnerabilities:
  - crashes, data corruption, developer productivity

AddressSanitizer (ASan)

AddressSanitizer

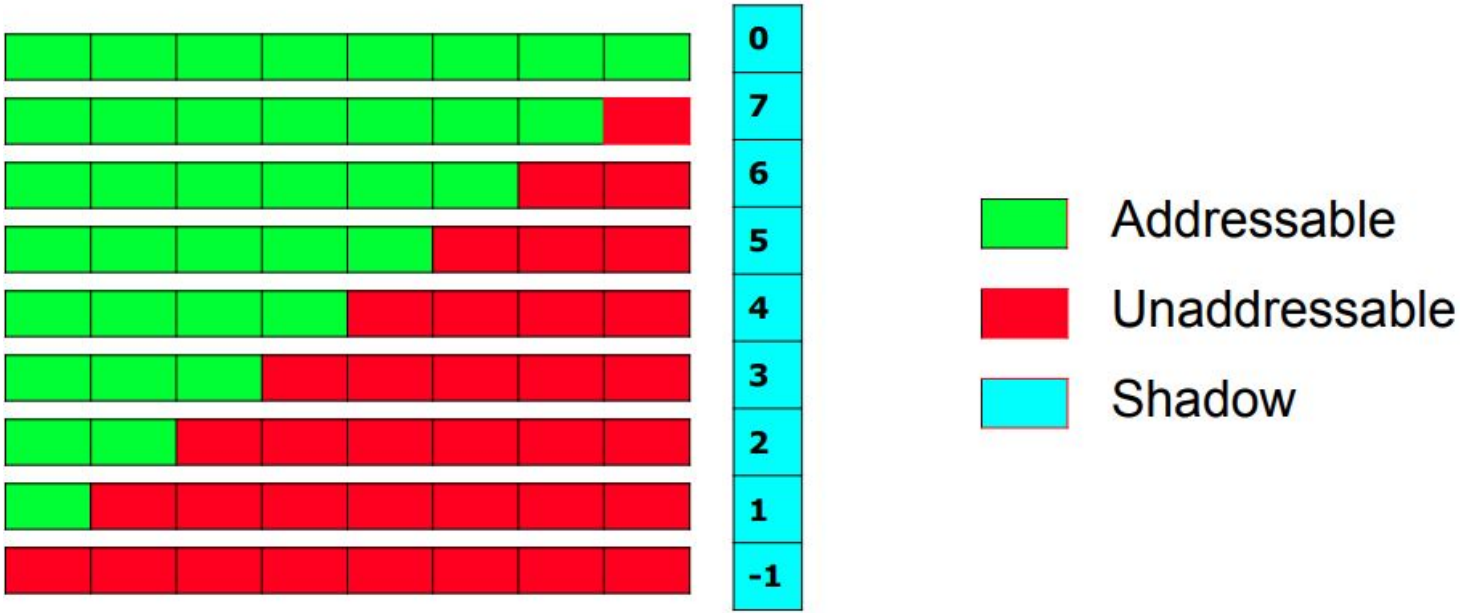
- Released in May 2011
- Memory error detector in C/C++ that finds most of the aforementioned bugs
- Compiler instrumentation + run-time library (that replaces malloc/free)
- Average slowdown: 2x, memory overhead: 1.5x-4x
- No false positives!
- During first 1.5 years found 1000+ bugs in:
  - Firefox, Chrome, Vim, GCC, MySQL, LLVM, Perl, FFmpeg, libjpeg-turbo, FreeType, ...
- `-fsanitize=address` (demo)

Mapping: `Shadow = (Addr>>3) + Offset`



Shadow byte

- Every aligned 8-byte word of memory has only 9 states
- First N bytes are addressable, the rest 8-N bytes are not
- Can encode in 1 byte (shadow byte)
- Extreme: 128 application bytes map to 1 shadow byte.



Instrumentation: 8 byte access

```
*a = ...
```

```
char *shadow = (a>>3)+Offset;
if (*shadow)

   ReportError(a);
*a = ...
```

Instrumentation: N byte access (N=1, 2, 4)

```
*a = ...
```

```
char *shadow = (a>>3)+Offset;
if (*shadow &&
     *shadow <= ((a&7)+N-1))
   ReportError(a);
*a = ...
```

AddressSanitizer (ASAN): redzones, quarantine

```
char *p = new char[24];
```


```
p[24] = ...  // OMG

delete [] p;  // kept in quarantine for a while
```


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

Instrumenting globals

```
int a;
```

```
struct {
   int original;
   char redzone[60];
} a;  // 32-aligned
```

Instrumenting stack

```
void foo() {
   char a[328];

   <-------------- CODE -------------->
}
```

Instrumenting stack

```
void foo() {
  char rz1[32];  // 32-byte aligned
  char a[328];
  char rz2[24];
  char rz3[32];
  int  *shadow = (&rz1 >> 3) + kOffset;
  shadow[0] = 0xffffffff;   // poison rz1

  shadow[11] = 0xffffff00;  // poison rz2
  shadow[12] = 0xffffffff;  // poison rz3
  <------------ CODE ------------>
  shadow[0] = shadow[11] = shadow[12] = 0;
}
```

ASan run-time library

- Initializes shadow memory at startup
- Provides full malloc replacement
  - Insert poisoned redzones around allocated memory
  - Quarantine for free-ed memory
  - Collect stack traces for every malloc/free
- Provides interceptors for functions like memset
- Prints error messages

The KernelAddressSanitizer

- Implementing ASan inside OS kernel is more difficult
- General idea is the same: shadow region + instrumented allocators (kernels usually have many different allocators)
- Compiler inserts `__asan_loadN()` and `__asan_storeN()` calls, that we have to implement
- Main challenge: setting up the shadow region early in the boot process (which is obviously machine-dependent)
- `-fsanitize=kernel-address`

The KernelAddressSanitizer

- Added to:
  - Linux (2014)
  - NetBSD (2018)
  - FreeBSD (2019)
  - Mimiker (2020)

ASan’s problems

- Hard to use in production (~2x overhead in memory/CPU/code size)
- Not a strong security mitigation:
  - buffer overflows: access may jump over redzone

```
char *a = new char[100];
char *b = new char[1000];
a[500] = 0;  // may end up somewhere in b
```

  - use-after-free: access may “outlive” quarantine

```
char *a = new char[1 << 20];  // 1MB
delete [] a;  // <<< "free"
char *b = new char[1 << 28];  // 256MB
delete [] b;  // drains the quarantine queue.
char *c = new char[1 << 20];  // 1MB
a[0] = 0;     // "use". May land in ’c’.
```

- Does not detect bugs in pre-compiled binaries

Memory tagging

Tagged architecture

- Type of computer architecture where every word of memory is a tagged union (i.e. data + tag)
- The idea is not new:
  - Rice Computer (ca. 1960)
  - Lisp machines (1970s, 1980s)

Memory tagging / coloring / tainting

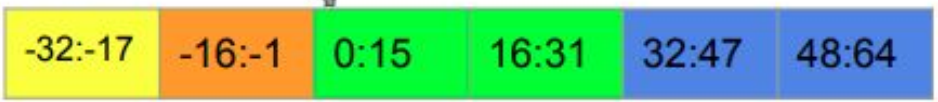
- 64-bit architectures only
- Every aligned TG bytes have a TS-bit tag
  - TG = tagging granularity, TS = tag size
  - Example values: (TG=64, TS=4) / (TG=16, TS=8)
- TS bits in the upper part of every pointer contain a tag

Memory tagging / coloring / tainting

- malloc():
  - align to Tagging Granularity
  - choose a tag (e.g. randomly)
  - tag the memory
  - tag the returned pointer
- free():
  - re-tag the memory
- Every load and store instruction raises an exception on mismatch between the pointer and memory tags

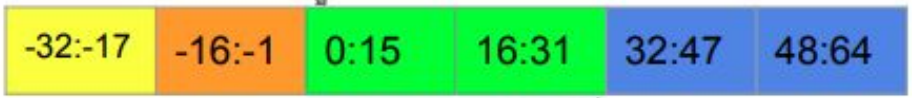
Heap-buffer-overflow

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



Heap-buffer-overflow

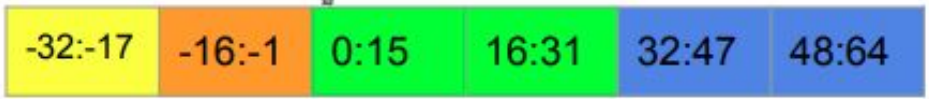
```
char *p = new char[20]; // 0xa007ffffffff1240
```




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

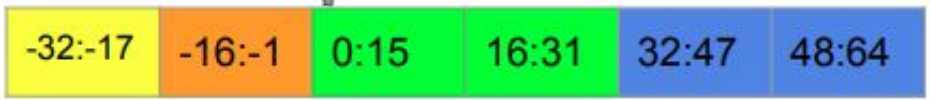
Heap-use-after-free

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

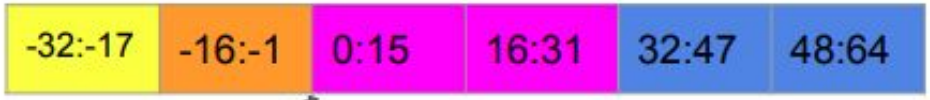


Heap-use-after-free

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




```
delete [] p;  // Memory is retagged ■ ⇒ ■
```




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

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, but requires redzone
  - 100% use-after-free-before-realloc
- Different tags for adjacent chunks
  - 100% off-by-one (linear) buffer-overflow detection
- ...

MT and stack-buffer-overflow, stack-use-after-return

- Compiler instrumentation still needed to tag/untag local variables
- Significant memory overhead for large TG values (imagine aligning stack variables to 64 bytes)
- Stack instrumentation can be optional

MT vs ASAN

- MT:
  - Expected smaller RAM overhead (no redzones, but some alignment)
  - Detection of buffer-overflows far from bounds
  - Detection of use-after-free long after deallocation (without quarantine)
  - Can initialize memory as a side effect
  - Requires some hardware support (we'll talk about it in a while)
- ASAN:
  - Precise 1-byte buffer-overflow detection
  - More portable (also for 32-bit, no need for hardware support)

MT vs ASAN

- Better detection of overflows and use-after-free-s makes MT much stronger security mitigation against attackers!

Existing implementations of MT

HWASAN (ASAN-MT hybrid)

- Hardware-assisted ASAN
- Only for AArch64 (which supports *top-byte-ignore*), we keep the tag in the MSB of a pointer
  - notice: x86-64/Aarch64 allow only 48-bit virtual address space
- Memory tags kept in the shadow region (shadow requires 1/TG extra memory)
- TG=16, TS=4
- Overhead: 2x CPU, 6% RAM, 2.5x code size

HWASAN (ASAN-MT hybrid)

```
// 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
```

HWASAN vs ASAN

- HWASAN:
  - probabilistic detection of errors, but detects far buffer-overflows and delayed use-after-free
  - much smaller memory overhead (no heap redzones, no quarantine, 2x smaller shadow region)
  - only for AArch64
  - rather expensive stack instrumentation (consider disabling it)
- Both have similar CPU overhead

SPARC ADI hardware extension

- ADI = Application Data Integrity
- Supported on SPARC M7/M8 CPUs since 2016
- TG=64, TS=4
- The memory tag for a single 64-byte region is set with a single instruction

STXA$^{P_{ASI}}$	01 1110	Store Extended Word into Alternate Space	`stxa` *reg$_{rd}$*, [*regaddr*] *imm_asi* `stxa` *reg$_{rd}$*, [*reg_plus_imm*] `%asi`		**A1**

- Load/store generates hardware exception on tag mismatch
- Memory tags:
  - not separately addressable
  - stored in some hidden hardware state
  - only way to read/write is through special instructions

SPARC ADI hardware extension

- Need to instrument malloc()/free(), but not each read/write in the application
  - CPU will detect errors itself
- Aligns malloc by 64
- Heap bugs only (costly stack instrumentation)

TLDR:

- Very low CPU overhead :-)
- Very low code size overhead :-)
- Only in SPARC! :-(

Incoming implementations of MT

ARM MTE (Memory Tagging Extension)

- MT for ARM announced at the end of 2018
- New hardware instructions for interacting with tagged memory, e.g.:

`GMI`	Tag Mask Insert	`GMI <Xd>, <Xn/SP>, <Xm>`
`IRG`	Insert Random Tag	`IRG <Xd/SP>, <Xn/SP>{, <Xm>}`
`LDG`	Load Allocation Tag	`LDG <Xt>, [<Xn/SP>{, #<simm>}]`
`LDGV`	Load Tag Vector	`LDGV <Xt>, [<Xn/SP>]!`
`ST2G`	Store Allocaton Tags	`ST2G [<Xn/SP>], #<simm>` `ST2G [<Xn/SP>, #<simm>]!`

`STG`	Store Allocation Tag	`STG [<Xn/SP>], #<simm>`
		`STG [<Xn/SP>, #<simm>]!`
		`STG [<Xn/SP>{, #<simm>}]`

ARM MTE (Memory Tagging Extension)

- TG=16 (so very little malloc overalignment), TS=4
- Doesn’t exist in hardware yet
- Similarly to SPARC ADI:
  - Load/store generates hardware exception on tag mismatch
  - Need only to instrument malloc()/free()
- Overheads:
  - RAM: 3-5%
  - CPU: “hoping for low-single-digit %”

Benefits of fully hardware-assisted MT

- Finds buggy accesses in non-instrumented code
  - e.g. some legacy code that we don’t want to recompile with ASAN
  - only changes in malloc()/free() required
- Could allow shipping instrumented binary to production
  - large overhead of ASAN makes it not feasible, so usually ASAN-build is used only for testing

Bonus: ThreadSanitizer

ThreadSanitizer v1

- Tool based on Valgrind that detects data races
- Released in 2009
- Slowdown 20-300x
  - still faster than other tools at that time
- Found thousands of races (at least 180 in Chromium)

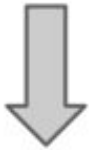
ThreadSanitizer v2

- Compiler instrumentation + run-time library
- Developed inside LLVM, imported into GCC
- `-fsanitize=thread` (demo)
- Slowdown 5-15x, memory overhead 5-10x
- It’s the current TSan

TSan detectable bugs

- Regular data races
- Races on mutexes, file descriptors, barriers, ...
- Use-after-free races
- Signal-unsafe malloc/free call in signal handler
- Initializing objects without synchronization
- Potential deadlocks
- ...

```
void foo(int *p) {
  *p = 42;
}
```



```
void foo(int *p) {
  __tsan_func_entry(__builtin_return_address(0));
  __tsan_write4(p);
  *p = 42;
  __tsan_func_exit()
}
```

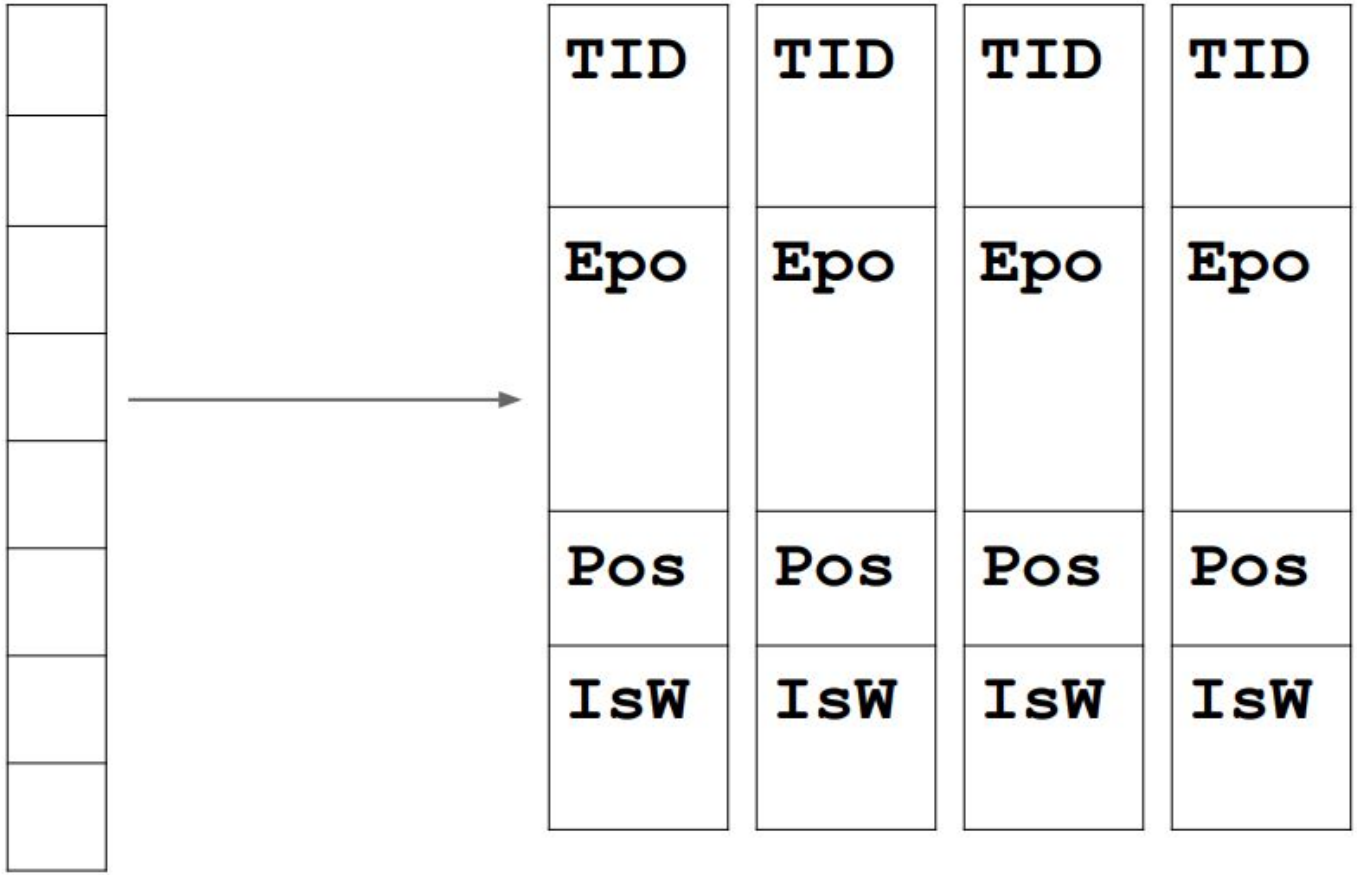
Shadow cell

An 8-byte shadow cell represents one memory access:

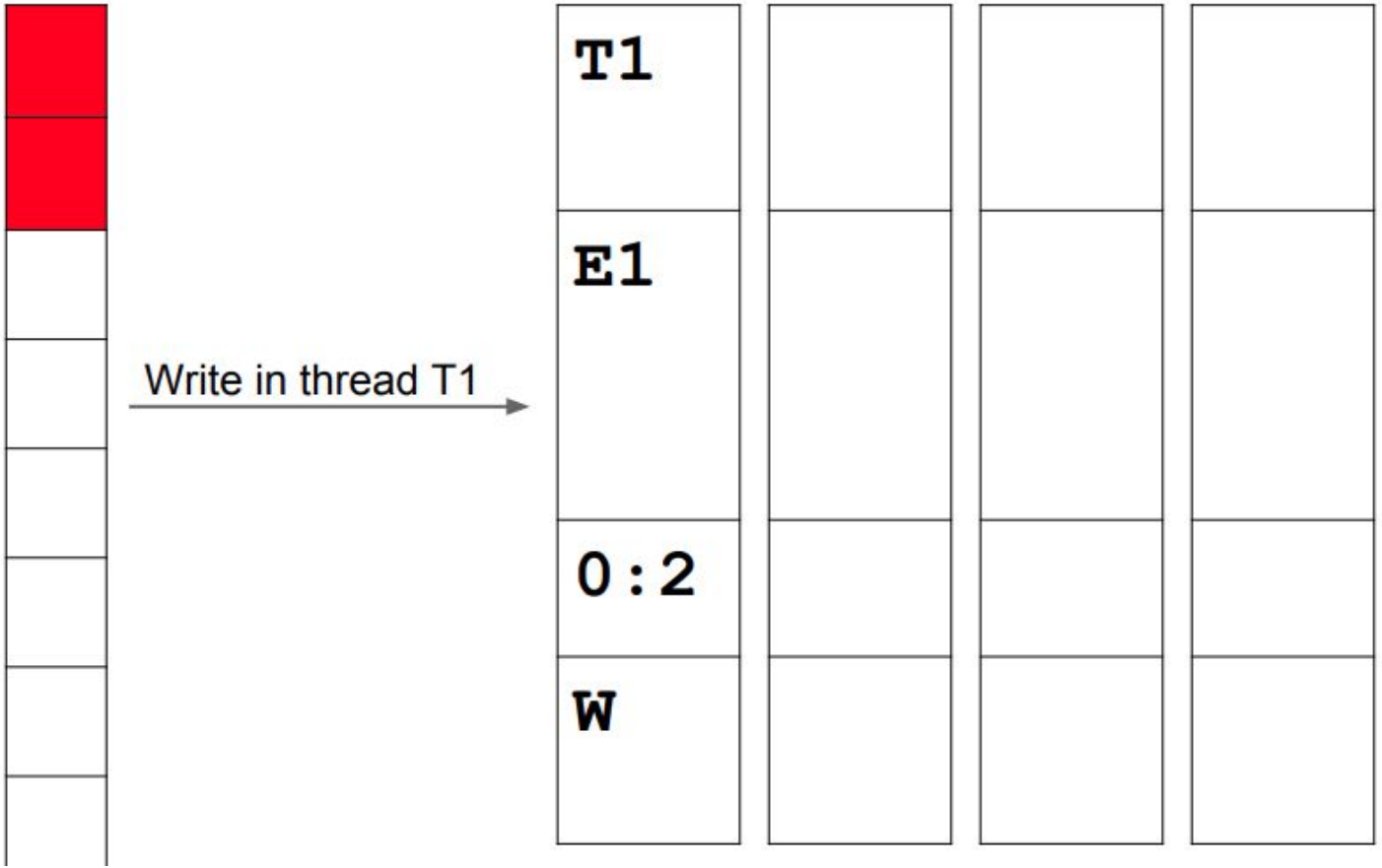
- ~16 bits: TID (thread ID)
- ~42 bits: Epoch (scalar clock)
- 5 bits: position/size in 8-byte word
- 1 bit: IsWrite

Full information (no more dereferences)

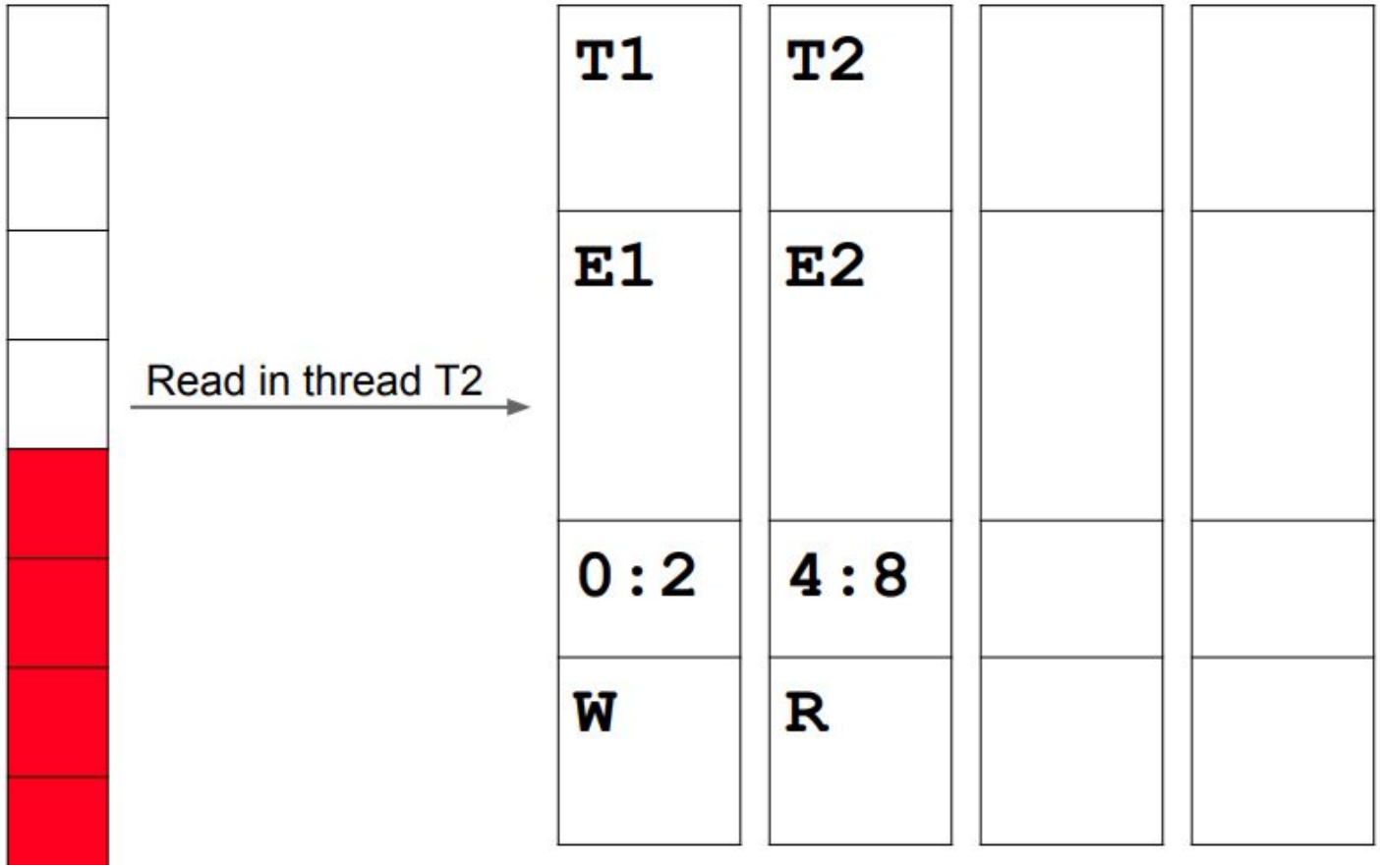
N shadow cells per 8 application bytes



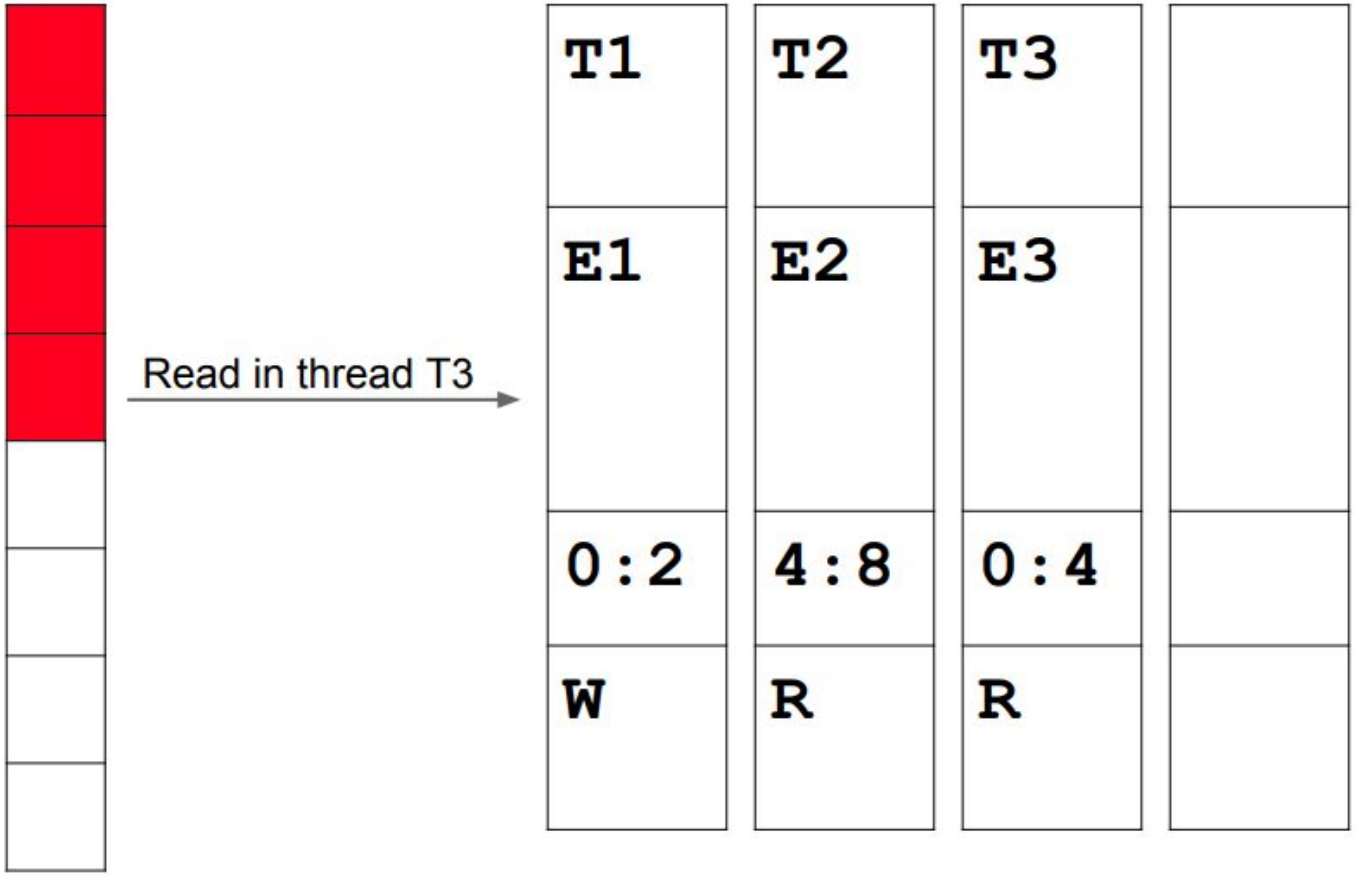
Example: first access



Example: second access



Example: third access



Example: race?

Race if **E1** not "happens-before" **E3**

T1	T2	T3	
E1	E2	E3	
0:2	4:8	0:4	
W	R	R	

Some overview of TSan v1 algorithm

- TSan v2 is implemented in a similar way
- We need to define *happens-before* relation
- TSan keeps track of: memory accesses, the aforementioned relation, mutex locking/unlocking, signal/wait events, and looks for a race condition
- Let’s look at an example (very simple) definition of *happens-before*
  - unfortunately it will sometimes report false-positives

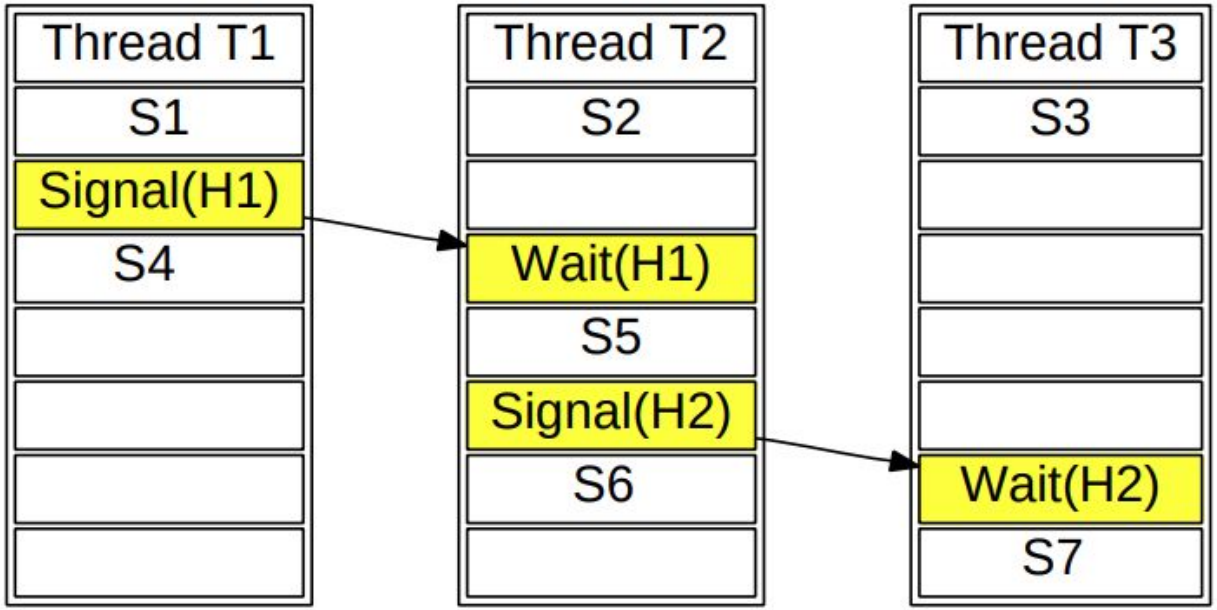


Figure 1: Example of happens-before relation. $S_1 \prec S_4$ (same thread); $S_1 \prec S_5$ (happens-before arc $\text{SIGNAL}_{T_1}(H_1) - \text{WAIT}_{T_2}(H_1)$); $S_1 \prec S_7$ (happens-before is transitive); $S_4 \not\prec S_2$ (no relation).

“Data race” definition

Concurrent: two memory access events X and Y are concurrent if $X \not\preceq Y$, $Y \not\preceq X$ and the intersection of the lock sets of these events is empty.

Data Race: a data race is a situation when two threads concurrently access a shared memory location (i.e. there are two *concurrent* memory access events) and at least one of the accesses is a WRITE.

“Event lock set” definition

Writer Lock Set (LS_{Wr}): the set of all write-held locks of a given thread.

Reader Lock Set (LS_{Rd}): the set of all read-held locks of a given thread.

Event Lock Set: LS_{Wr} for a WRITE event and LS_{Rd} for a READ event.

Better “happens-before” definition

- Current *happens-before* looks only at SIGNAL -> WAIT pairs (where SIGNAL is observed first)
- Better definitions could also look at another pairs of events:
  - RD-UNLOCK -> WR-LOCK,
  - WR-UNLOCK -> RD-LOCK,
  - …
- Fewer false-positives!

Shadow cell eviction

- When all shadow cells are filled, one random is replaced

TSan function interceptors

Over 100 interceptors:

- malloc, free, …
- pthread_mutex_lock, …
- strlen, memcmp, …
- read, write, ...

Informative reports

- Need to report two stack traces:
  - current (easy)
  - previous (hard)
- Previous stack trace in TSan v2:
  - Per-thread cyclic buffer of events
    - event: memory access, function entry/exit
  - Information will be lost after some time

Bibliography

- **Memory Tagging and how it improves C/C++ memory safety.** K. Serebryany et al. 2018.
- **ThreadSanitizer – data race detection in practice.** K. Serebryany et al. 2009.
- https://www.youtube.com/watch?v=lLEcbXidK2o/
- https://github.com/google/sanitizers/
- https://gcc.gnu.org/wiki/cauldron2012
- https://lwn.net/Articles/598486/
- https://msrnd-cdn-stor.azureedge.net/bluehat/bluehatil/2019/assets/doc/Trends%2C%20Challenges%2C%20and%20Strategic%20Shifts%20in%20the%20Software%20Vulnerability%20Mitigation%20Landscape.pdf
- https://en.wikipedia.org/wiki/Tagged_architecture