Forewarned is forearmed: AddressSanitizer & ThreadSanitizer

Timur Iskhodzhanov, Alexander Potapenko,
Alexey Samsonov, Kostya Serebryany,
Evgeniy Stepanov, Dmitry Vyukov

November 2012

Agenda

- AddressSanitizer
  - detects use-after-free and buffer overflows (C++)
- ThreadSanitizer
  - detects data races (C++ & Go)
- Demos
- Practice

AddressSanitizer

use-after-free and buffer overflows

Everything is in C/C++

(even if you don't notice that):

- VMs (Java, Perl, Python)
- DBs (MySQL, PostgreSQL)
- Web servers (Apache, nginx)
- Web browsers (Chrome, Firefox, Safari)
- Everything else (libpng, libz, libxyz)

Why C/C++?

Efficient memory management +
Proximity to hardware =
  Speed

Hard to program +
Hard to debug =
Memory errors -- open gates to hackers

Debugging memory issues

Binary instrumentation:

- Valgrind, Dr. Memory, Intel Parallel Studio, Purify, Bounds Checker, Insure++, ...
- Veeery slow (> 20x), heap bugs only

Debug malloc:

- Page-level protection (efence, libgmalloc, Page Heap)
- Magic values
- Inaccurate (may miss bugs), slow, heap only

AddressSanitizer (ASan)

Instrumenting compiler + a runtime library

A bit of history:

- May 2011: v. 0.0
- May 2012: part of LLVM 3.1
- upcoming support in GCC 4.8

```
clang -fsanitize=address foo.c
```

Works on Linux, Mac OS, Android

AddressSanitizer

- buffer overflows
  - Heap
  - Stack
  - Global objects
- use-after-free, use-after-return
- double-free, memcpy-param-overlap etc.

ASan vs Valgrind vs debug malloc

	Valgrind	*-malloc	ASan
Heap out-of-bounds	YES	Sometimes	YES
Stack out-of-bounds	NO	NO	YES
Global out-of-bounds	NO	NO	YES
Use-after-free	YES	YES	YES
Use-after-return	NO	NO	Sometimes
Uninitialized reads	YES	NO	NO
CPU Overhead	10x-300x	??	1.5x-3x

Report example: use-after-free

```
ERROR: AddressSanitizer heap-use-after-free
  on address 0x7fe8740a6214
  at pc 0x40246f bp 0x7fffe5e463e0 sp 0x7fffe5e463d8

READ of size 4 at 0x7fe8740a6214 thread T0
    #0 0x40246f in main example_UseAfterFree.cc:4
    #1 0x7fe8740e4c4d in __libc_start_main ??:0

0x7fe8740a6214 is located 4 bytes inside of 400-byte
region

freed by thread T0 here:
    #0 0x4028f4 in operator delete[](void*) _asan_rtl_
    #1 0x402433 in main example_UseAfterFree.cc:4

previously allocated by thread T0 here:
    #0 0x402c36 in operator new[](unsigned long)
_asan_rtl_
    #1 0x402423 in main example_UseAfterFree.cc:2
```

Example: stack-buffer-overflow

```
ERROR: AddressSanitizer stack-buffer-overflow
  on address 0x7f5620d981b4
  at pc 0x4024e8 bp 0x7fff101cbc90 sp 0x7fff101cbc88

READ of size 4 at 0x7f5620d981b4 thread T0
    #0 0x4024e8 in main example_StackOutOfBounds.cc:4
    #1 0x7f5621db6c4d in __libc_start_main ??:0
    #2 0x402349 in _start ??:0

Address 0x7f5620d981b4 is located at offset 436 in frame
<main>
of T0's stack:
  This frame has 1 object(s):
    [32, 432) 'stack_array'
```

Trophies

- Chromium (including Webkit)
- Hundreds of bugs within Google
- Firefox
- FreeType, FFmpeg, WebRTC, libjpeg-turbo
- Perl, LLVM, GCC
- MySQL
- mod_rails
- even VIM and Git!

Which of the above do you use?

Fun facts

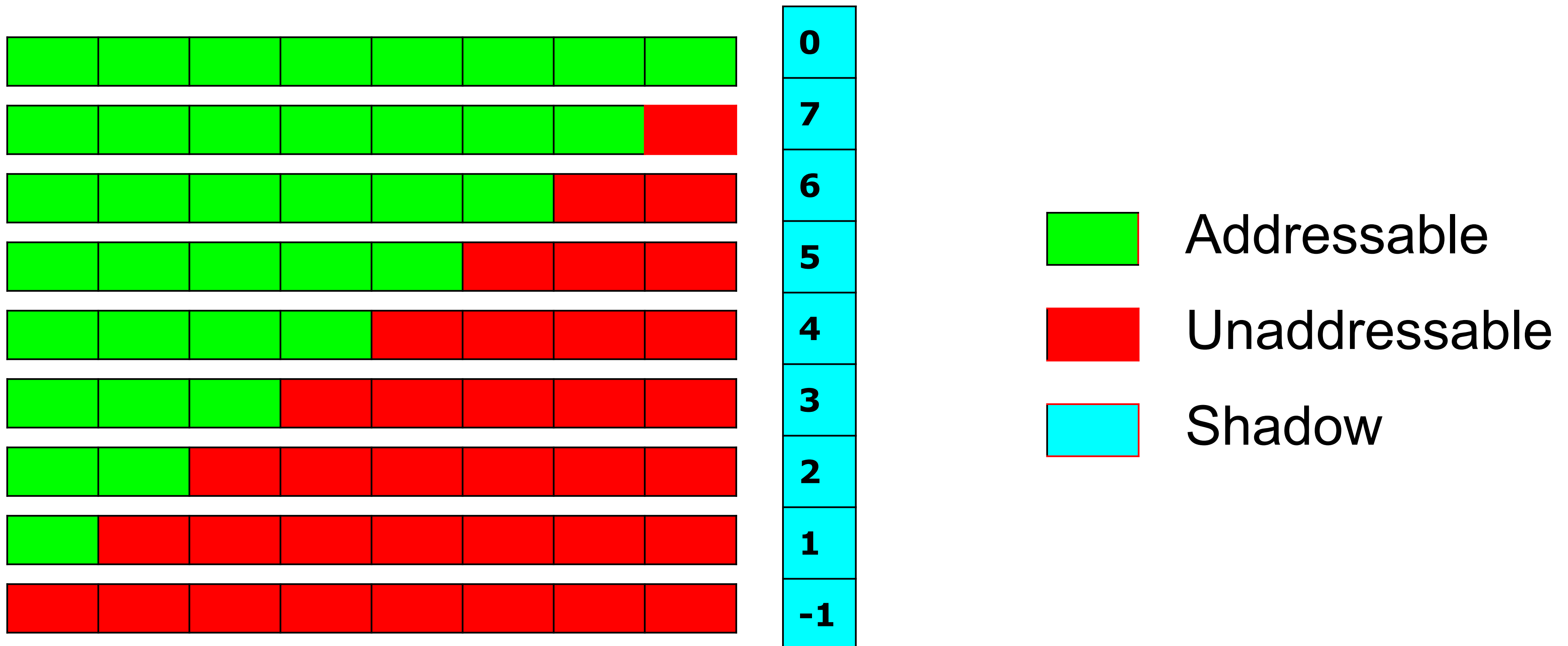
Google paid > $130K to external security researchers using ASan (like attekett and miaubiz)

One of the bugs Pinkie Pie exploited during the last Pwnium was in fact detectable with ASan

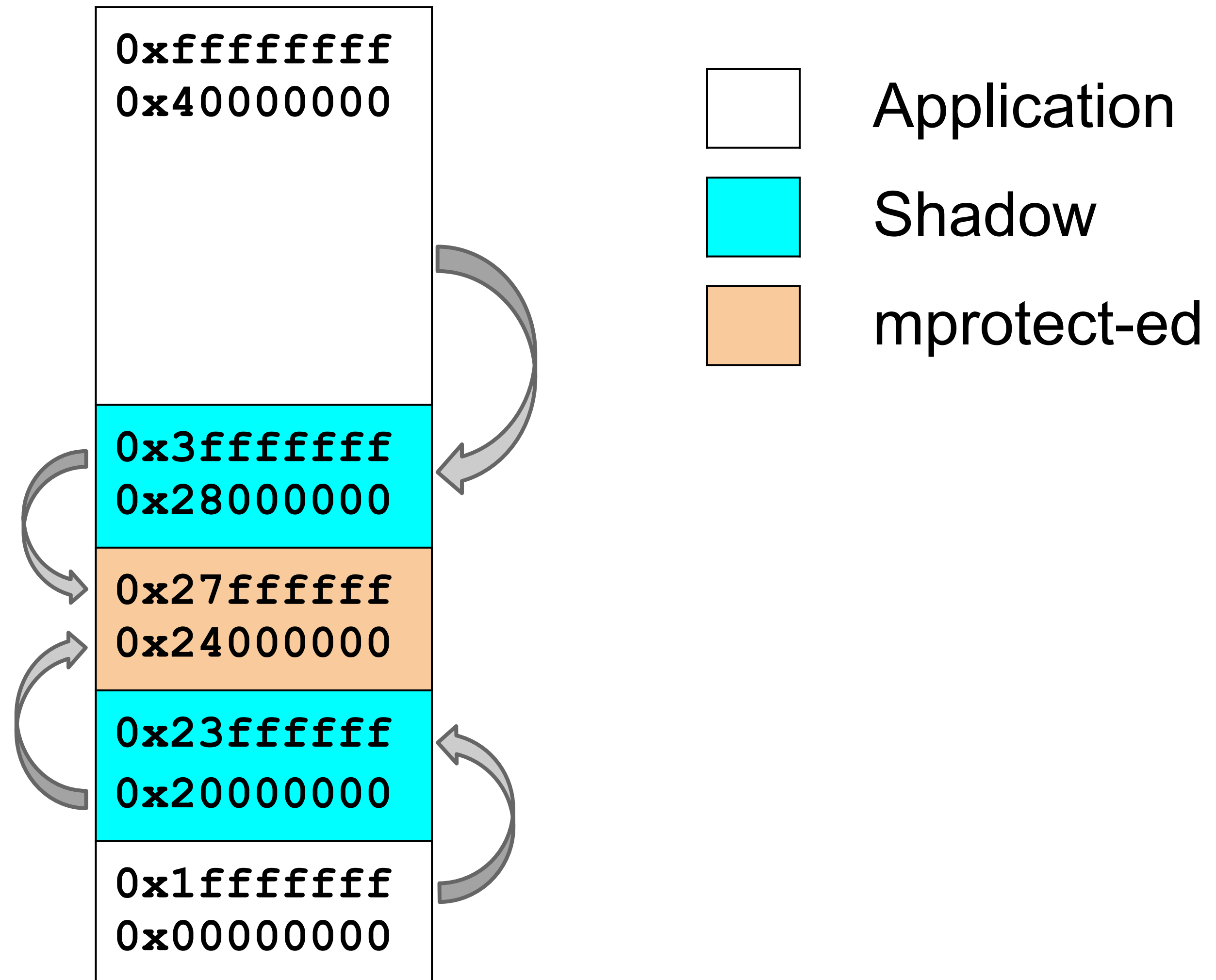
AddressSanitizer
How does it work?

Shadow byte

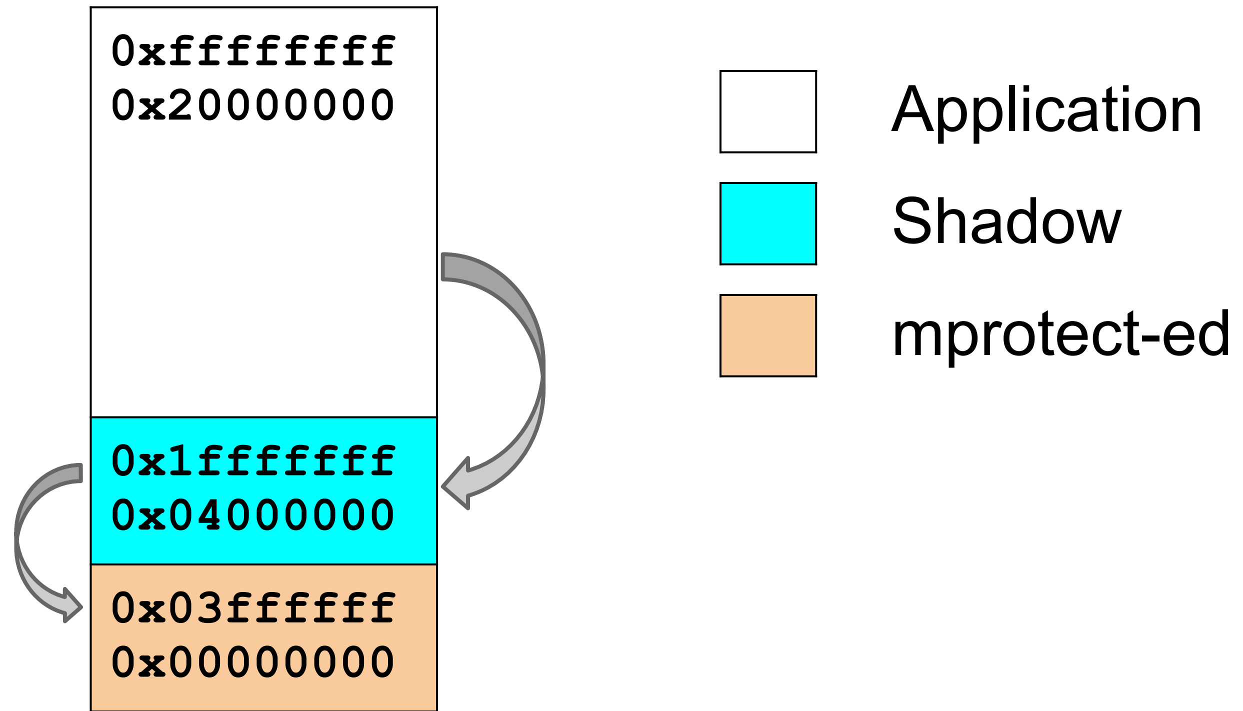
- Every aligned 8-byte word of memory has only 9 states
- First N bytes are addressable, the rest 8-N bytes are not
- Can encode in 1 byte (shadow byte)
- Extreme: 128 application bytes map to 1 shadow byte.



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



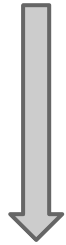
Mapping: `Shadow = (Addr>>3) + 0`



- Requires **`-fPIE -pie`** (linux)
- Gives ~6% speedup on x86_64

Instrumentation: 8 byte access

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

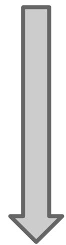


```
char *shadow = (a>>3)+Offset;
if (*shadow)

    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 = ...
```

Instrumentation example (x86_64)

```
shr $0x3,%rax                  # shift by 3
mov $0x100000000000,%rcx
or %rax,%rcx                   # add offset
cmpb $0x0,(%rcx)               # load shadow
je 1f <foo+0x1f>               # check for zero
call 409920                    # __asan_report
movq $0x1234,(%rdi)            # original store
```

Instrumenting stack

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



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

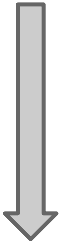
Instrumenting stack

```
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;
}
```

Instrumenting globals

```
int a;
```



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

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

Performance

- 1.7x on benchmarks (SPEC CPU 2006)
- Almost no slowdown for GUI programs
    - Chrome, Firefox
    - They don't consume all of CPU anyway
- 1.5x–4x slowdown for server side apps with -O2
    - The larger the slower (instruction cache)

Memory overhead: 2x–4x

- Redzones
  - Heap: 128–255 bytes / allocation
  - Global: 32–63 bytes / global var
  - Stack: 32–63 bytes / addr-taken local var (stack size increased up to 3x)
- Fixed size quarantine (256M)

- Shadow:
  - `mmap(MAP_NORESERVE)` 1/8-th of all address space
    - 16T on 64-bit
    - 0.5G on 32-bit
  - not more than (Heap + Globals + Stack + Quarantine) / 8

Ok, we have a hammer. Now what?

Run the tests

The more the better.

ASan @Chrome

Pre-submit trybots (optional)
Buildbots checking the new commits
ClusterFuzz -- 50,000,000 tests/day
Special "canary" Chrome build

Canarizing the desktop app

Aw, Snap!

Something went wrong while displaying this webpage. To continue, reload or go to another page.

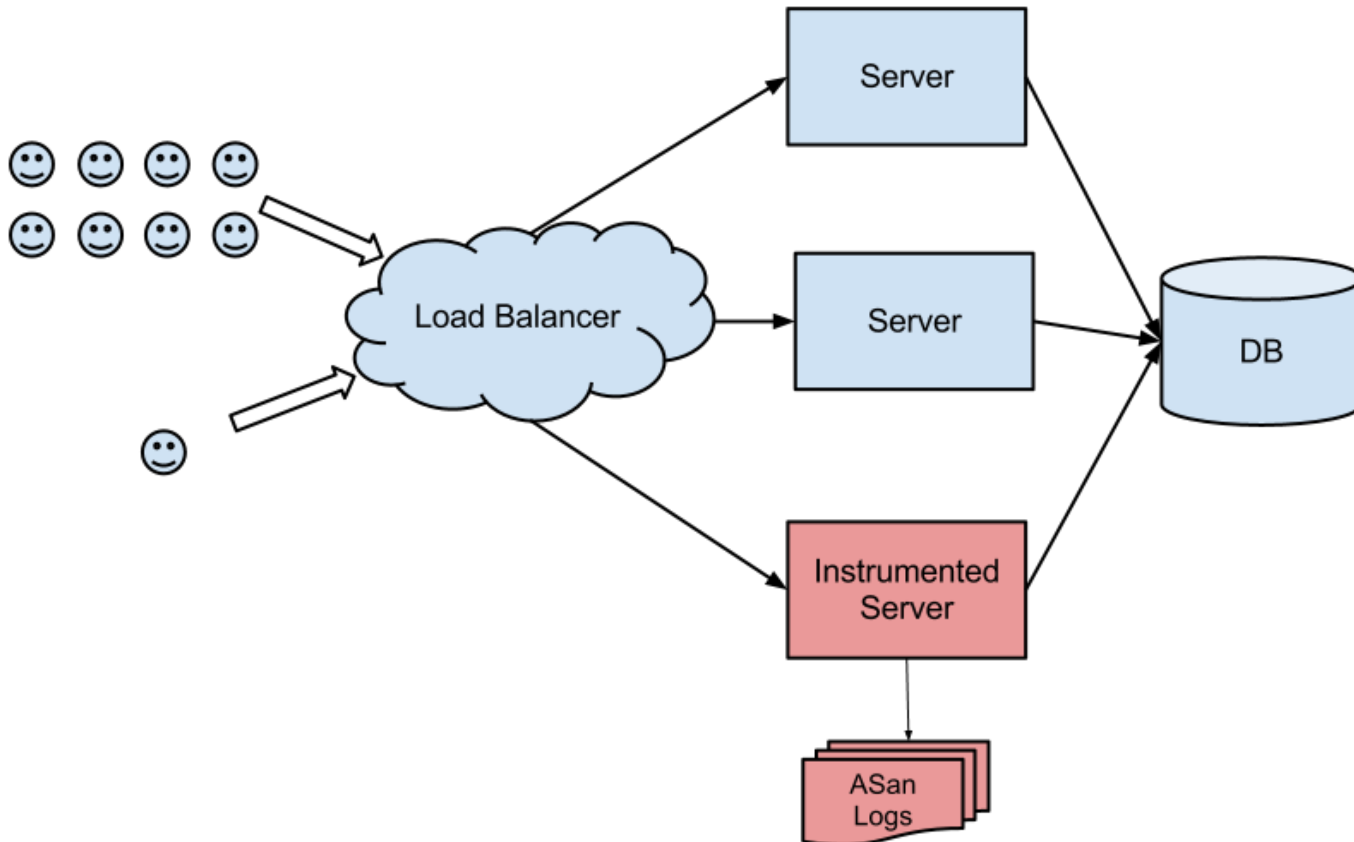
If you're seeing this frequently, try these suggestions.

-1-day bug detection

glider: Of course there're plenty of ClusterFuzz samples that we can use to show that ASan can detect problems in Chrome. But maybe there've been any actual exploits for problems detectable with ASan?

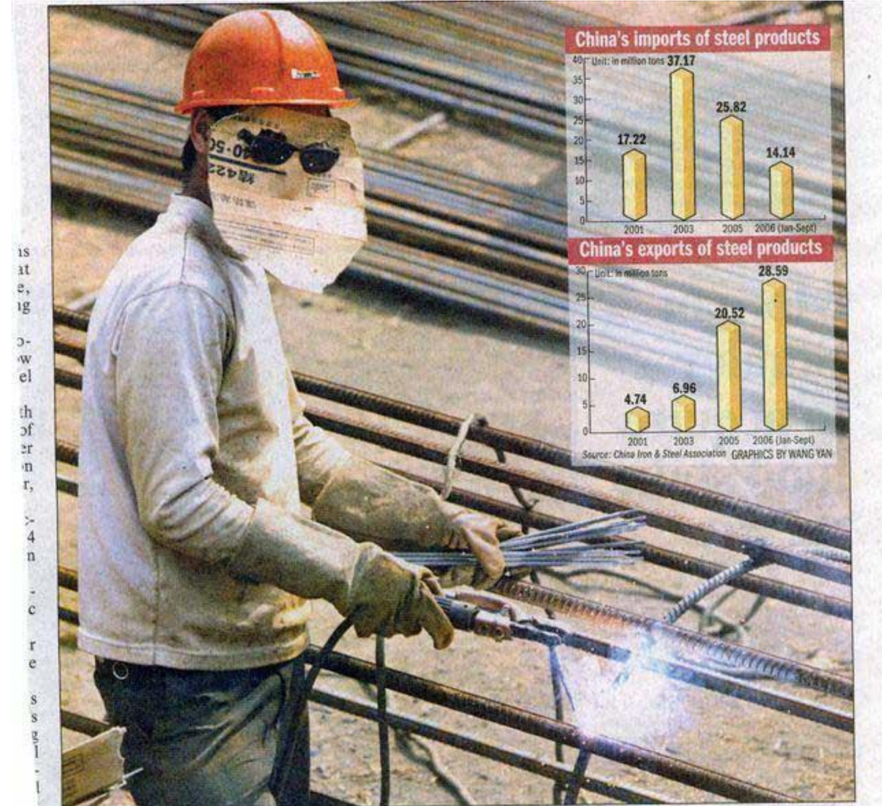
cevans: Nothing springs to mind. Because of a combination of ASan, ClusterFuzz, our reward program, autoupdate and a culture of prompt bug fixing, we tend to find the bugs first, so no actual exploits.

Canarizing the production



Honeypot?

Not sure. Maybe.
I'm not a real welder.



A worker welds at a construction site in Nanjing, the capital of East China's Jiangsu Province. FILE PHOTO

http://johns-jokes.com/afiles/images/saw_014_welding_mask.jpg

A poor man's sandbox

- bearable slowdown
  - < 2x on average
  - 3x–3.5x on large binaries (instruction cache)

- instant crashes on memory errors
  - a bit annoying for desktop users
  - good for server-side (though DoS still possible)

Is it safer?

ASan is a blackbox (code and heap layout are uncommon)

UAF/UAR

- need to exhaust the quarantine (250M)

Buffer overflow

- need to break the shadow protection somehow
- or exploit an overflow in the library code

Still vulnerable – we can do better

Not (yet) instrumented:

- JITted code
- inline assembly
- syscalls
- library routines
  - sprintf() is still a problem
  - wrappers to the rescue

No redzones in PODs

Little randomization

Future work

- Avoid redundant checks (static analysis)
- Instrument or recompile libraries
- Instrument inline assembly
- Adapt to use in a kernel
- Port to Windows
  - Mostly, frontend work (run-time works)
  - Plain C and simple C++ already works
  - Help is welcome!

ThreadSanitizer

data races

ThreadSanitizer v1

- Based on Valgrind
- Used since 2009
- Slow (20x–300x slowdown)
  - Found thousands races
  - Faster than others
    - Helgrind (Valgrind)
    - Intel Parallel Inspector (PIN)

ThreadSanitizer v2 overview

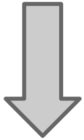
- Simple compile-time instrumentation
- Redesigned run-time library
  - Fully parallel
  - No expensive atomics/locks on fast path
  - Scales to huge apps
  - Predictable memory footprint
  - Informative reports

Slowdown

Application	Tsan1	Tsan2	Tsan1/Tsan2
RPC benchmark	283x	8.5x	33x
Server app test	28x	2x	14x
String util test	30x	2.4x	13x

Compiler instrumentation

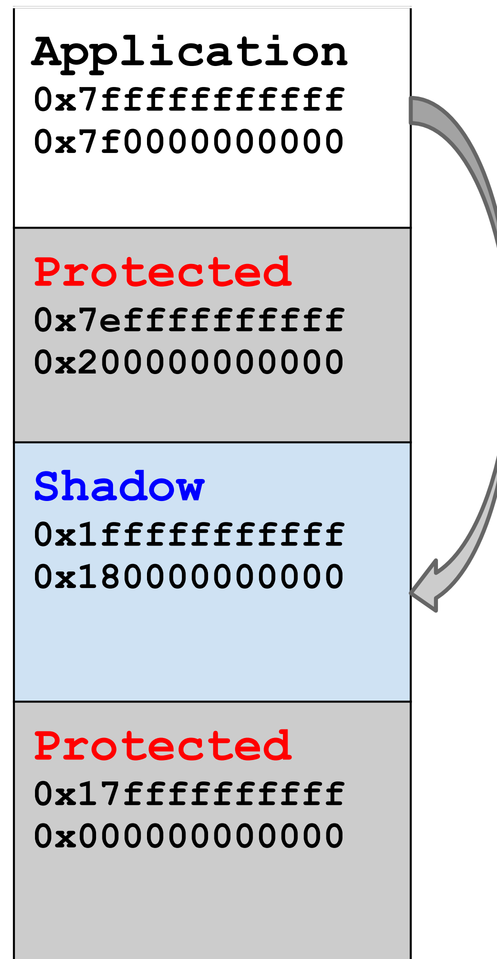
```
void foo(int *p) {
  *p = 42;
}
```



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

Direct mapping (64-bit Linux)

`Shadow = N * (Addr & kMask);` // Requires -pie

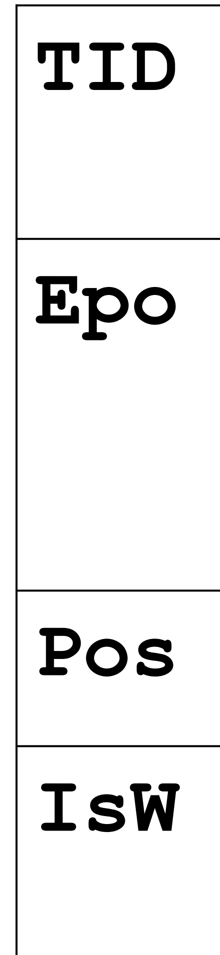


Shadow cell

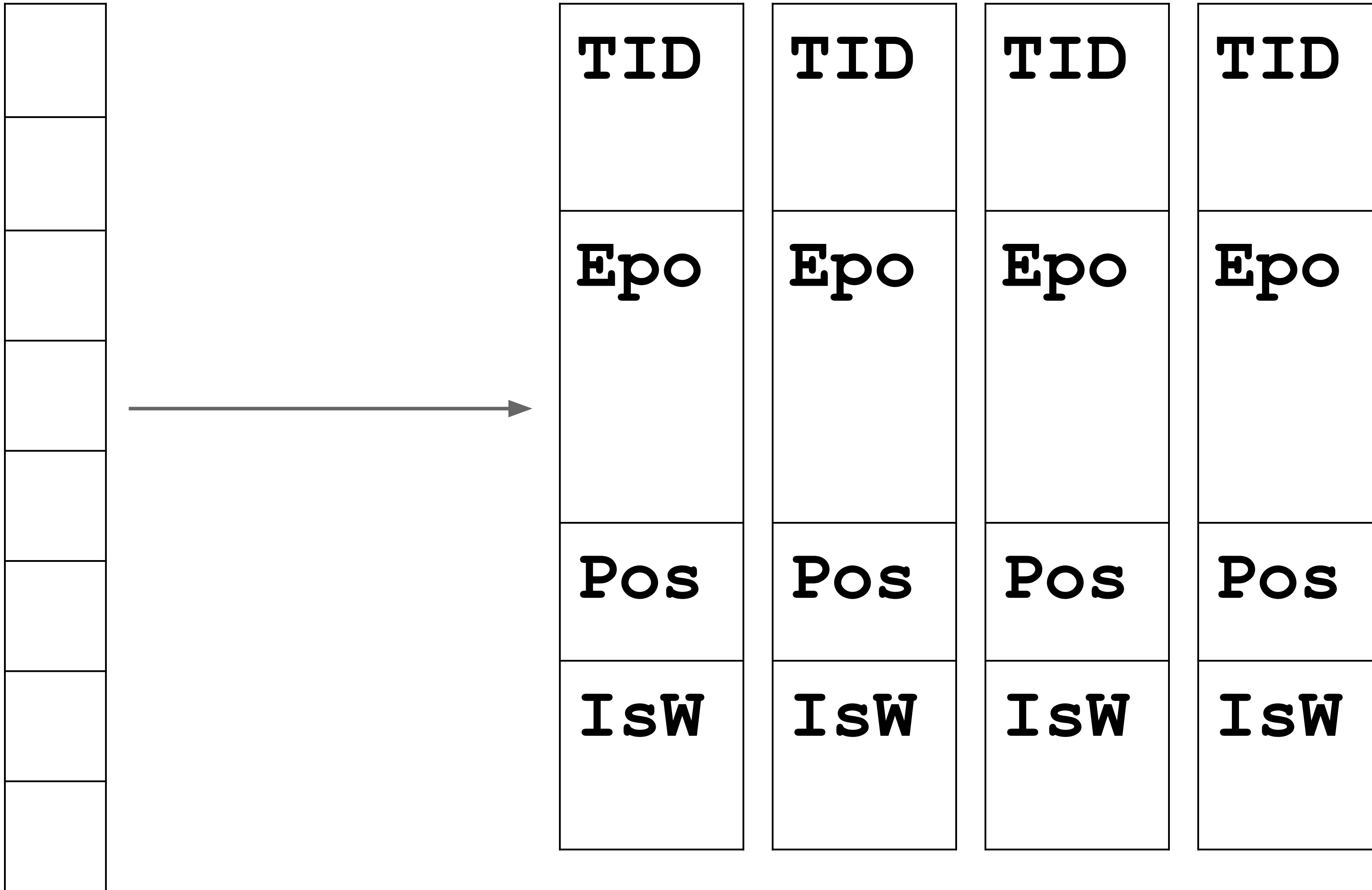
An 8-byte shadow cell represents one memory access:

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

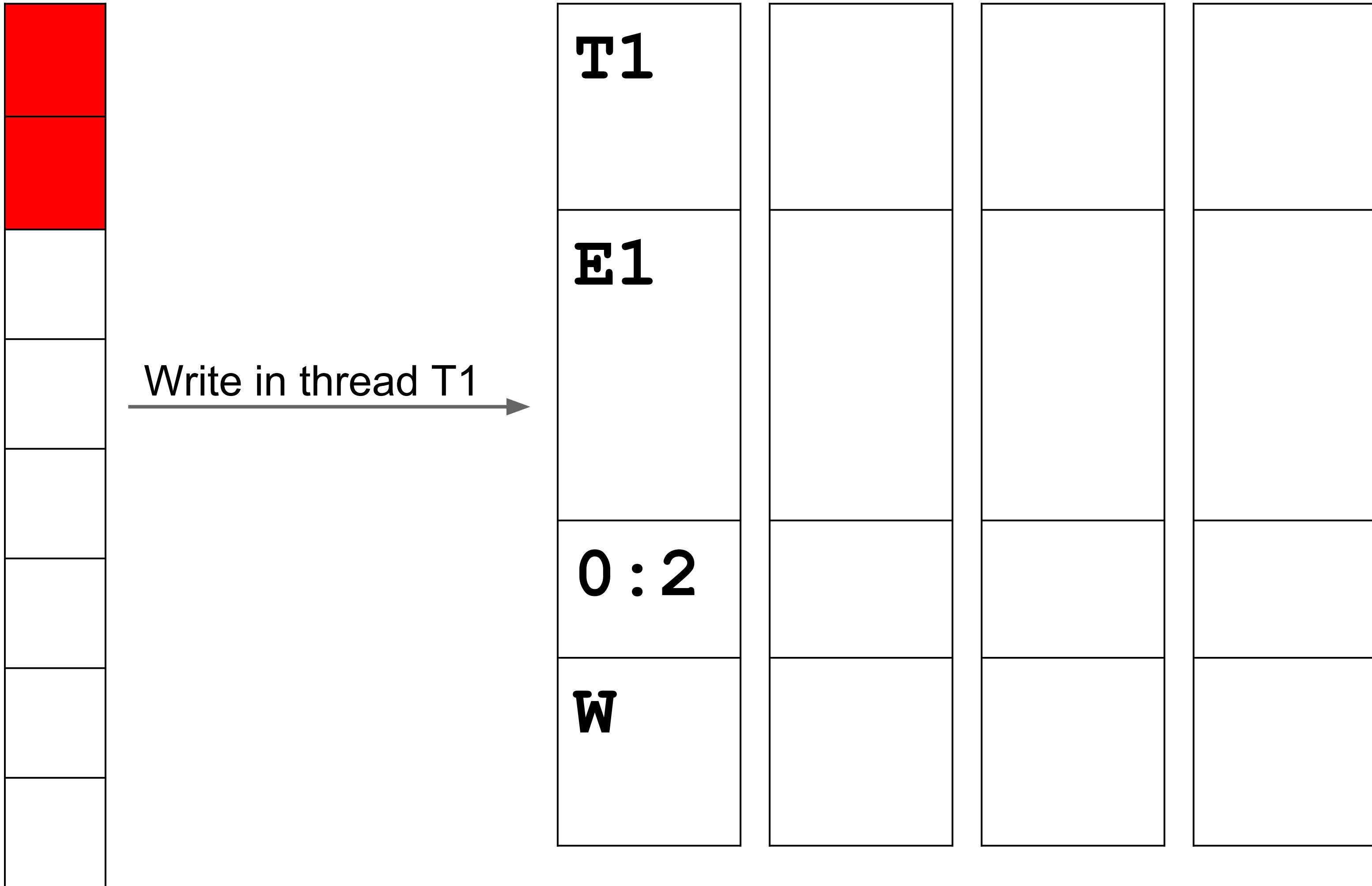
Full information (no more dereferences)



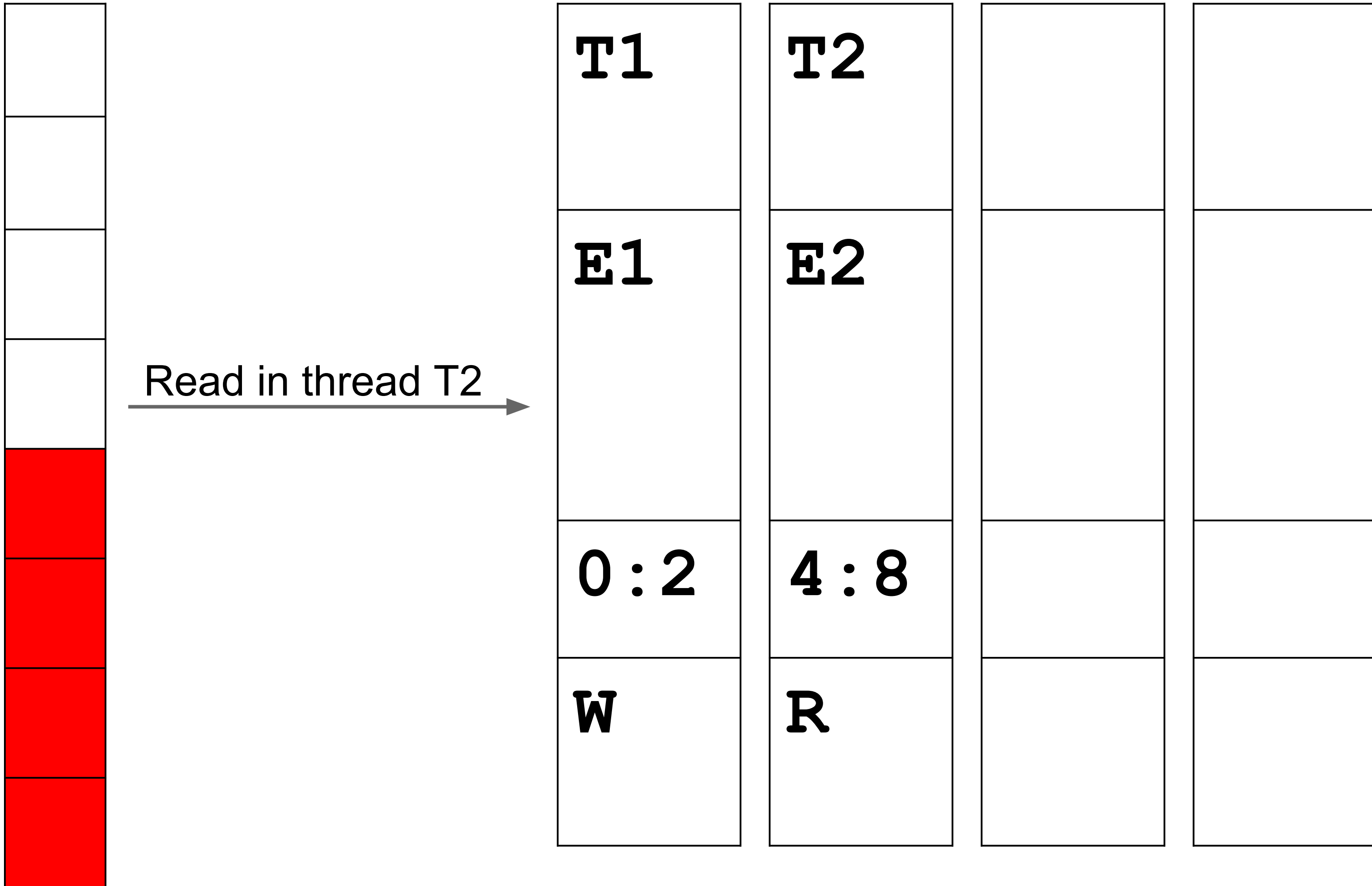
N shadow cells per 8 application bytes



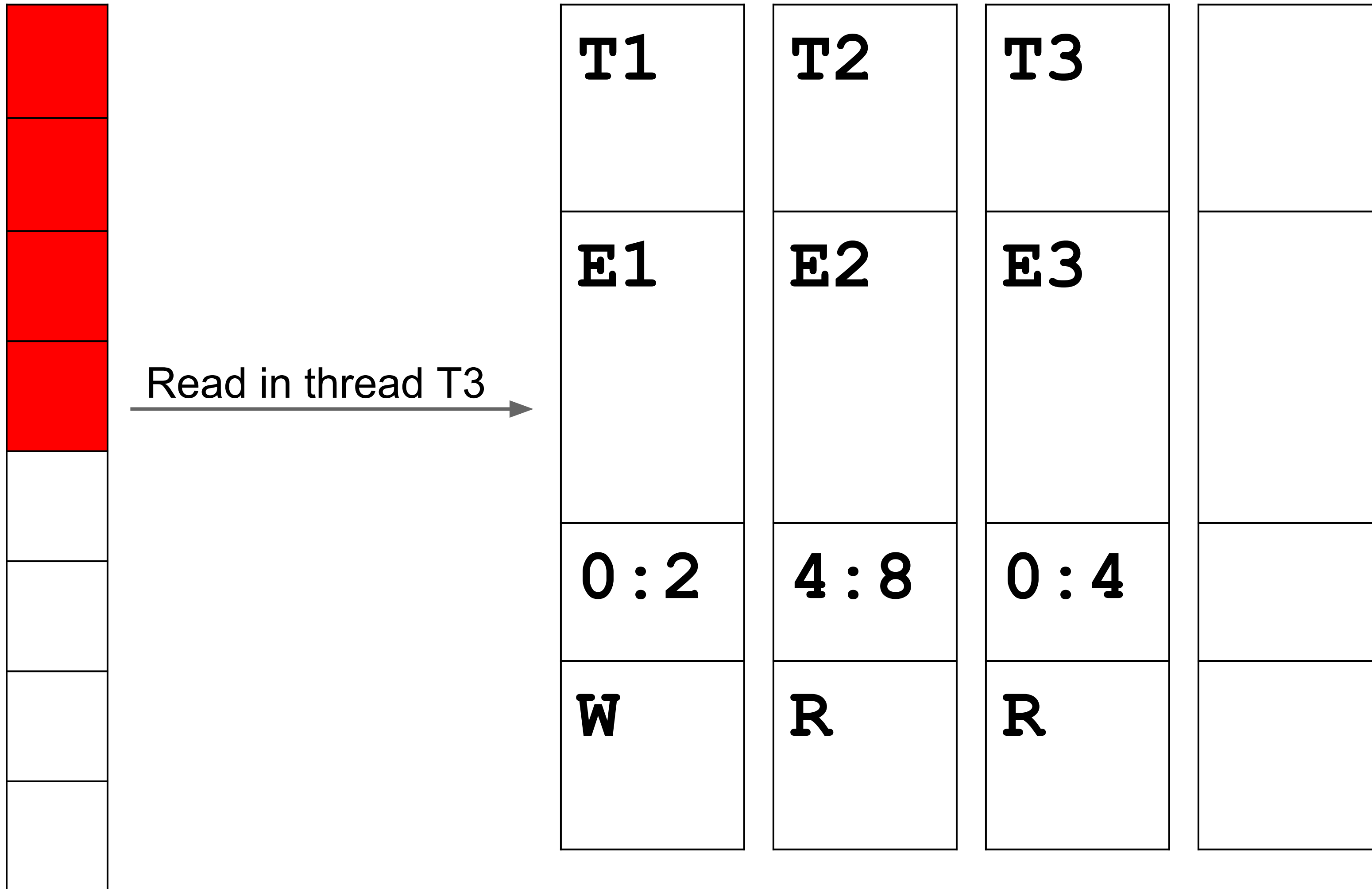
Example: first access



Example: second access



Example: third access



Example: race?

Race if **E1** does not "happen-before" **E3**

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

Fast happens-before

- Constant-time operation
  - Get TID and Epoch from the shadow cell
  - 1 load from thread-local storage
  - 1 comparison
- Similar to FastTrack (PLDI'09)

Shadow word eviction

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

Informative reports

- Need to report stack traces for two memory accesses:
  - current (easy)
  - previous (hard)

- TSan1:
  - Stores fixed number of frames (default: 10)
  - Information is never lost
  - Reference-counting and garbage collection

Stack trace for previous access

- Per-thread cyclic buffer of events
  - 64 bits per event (type + PC)
  - Events: memory access, function entry/exit
  - Information will be lost after some time
- Replay the event buffer on report
  - Unlimited number of frames

Function interceptors

- 100+ interceptors
  - malloc, free, ...
  - pthread_mutex_lock, ...
  - strlen, memcmp, ...
  - read, write, ...

Trophies

- 200+ bugs in Google server-side apps (C++)
- 100+ bugs in Go programs
  - 30+ bugs in Go stdlib
- Several races in OpenSSL
  - 1 fixed, ~5 'benign'

Limitations

- Only 64-bit Linux
- Heavily relies on TLS
  - Slow TLS on some platforms
- Hard to port to 32-bit platforms :(
  - Too small address space
  - Expensive atomic 64-bit load/store
- Does not instrument:
  - pre-built libraries
  - inline assembly

Demo time

AddressSanitizer

Stack use-after-return -- common in large programs with callbacks, very hard to debug.

Stack buffer overflow -- can sometimes be caught by stack protectors.

ASan detects both easily.

ThreadSanitizer

Thread-**un**safe reference counting =>
=> execution of malicious code.

Basically impossible to debug.

TSan finds instantly, explains, helps to verify fix.

$ clang **-fsanitize=thread**

Practice

`bug-list.txt` -- list of bugs in the tests

`fuzz.sh` -- run all the tests under ASan/TSan

`test*.cc` -- tests accepting a 2-char argument, some arguments trigger bugs in the tests.

```
$ clang -fsanitize=address -g t.c
$ ./a.out aa 2>& | ./asan_symbolize.py
$ clang -fsanitize=thread -fPIE -pie -g t.c
$ ./a.out aa
```

Task: map test numbers to bug descriptions

Q&A

http://code.google.com/p/address-sanitizer/

http://code.google.com/p/thread-sanitizer/

{glider,dvyukov}@google.com