ThreadSanitizer, MemorySanitizer

Scalable run-time detection of uninitialized memory reads and data races with LLVM instrumentation

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

LLVM developers' meeting, Nov 8 2012

Agenda

- AddressSanitizer (aka ASan)
  - recap from 2011
  - detects use-after-free and buffer overflows (C++)
- ThreadSanitizer (aka TSan)
  - detects data races (C++ & Go)
- MemorySanitizer (aka MSan)
  - detects uninitialized memory reads (C++)
- Similar tools, find different kinds of bugs

AddressSanitizer (recap from 2011)

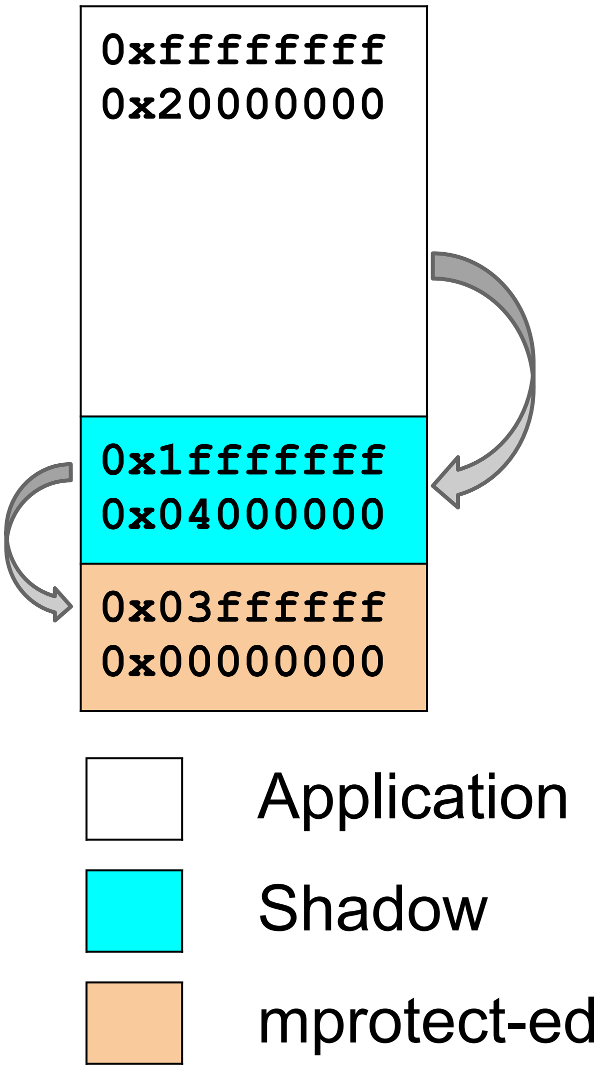
- Finds
  - buffer overflows (stack, heap, globals)
  - use-after-free
  - some more
- LLVM compiler module (~1KLOC)
  - instruments all loads/stores
  - inserts red zones around Alloca and GlobalVariables
- Run-time library (~10KLOC)
  - malloc replacement (redzones, quarantine)
  - Bookkeeping for error messages

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
```

ASan shadow memory

Virtual address space



Instrumentation

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

```
char *shadow
 = addr >> 3;
if (*shadow)
  ReportError(a);
*a = ...
```

ASan *marketing* slide

- 2x slowdown (Valgrind: 20x and more)
- 1.5x-4x memory overhead
- 500+ bugs found in Chrome in 1.5 years
  - Used for tests and fuzzing, 2000+ machines 24/7
  - 100+ bugs by external researchers
- 1000+ bugs everywhere else
  - Firefox, FreeType, FFmpeg, WebRTC, libjpeg-turbo, Perl, Vim, LLVM, GCC, MySQL

Plea to hardware vendors

Trivial hardware support may reduce the overhead from 2x to 20%

ThreadSanitizer

data races

ThreadSanitizer v1

- Race detector based on Valgrind
- Used since early 2009
- Slow (20x–300x slowdown)
  - Still, found thousands races
  - Faster & more usable than others
    - Helgrind (Valgrind)
    - Intel Parallel Inspector (PIN)
- WBIA'09

ThreadSanitizer v2 overview

- Simple compile-time instrumentation
  - ~400 LOC
- Redesigned run-time library
  - Fully parallel
  - No expensive atomics/locks on fast path
  - Scales to huge apps
  - Predictable memory footprint
  - Informative reports

TSan report example: data race

```
void Thread1() { Global = 42; }
int main() {
  pthread_create(&t, 0, Thread1, 0);
  Global = 43;
  ...
% clang -fsanitize=thread -g a.c -fPIE -pie && ./a.out
WARNING: ThreadSanitizer: data race (pid=20373)
  Write of size 4 at 0x7f... by thread 1:
    #0 Thread1 a.c:1
  Previous write of size 4 at 0x7f... by main thread:
    #0 main a.c:4
  Thread 1 (tid=20374, running) created at:
    #0 pthread_create ??:0
    #1 main a.c:3
```

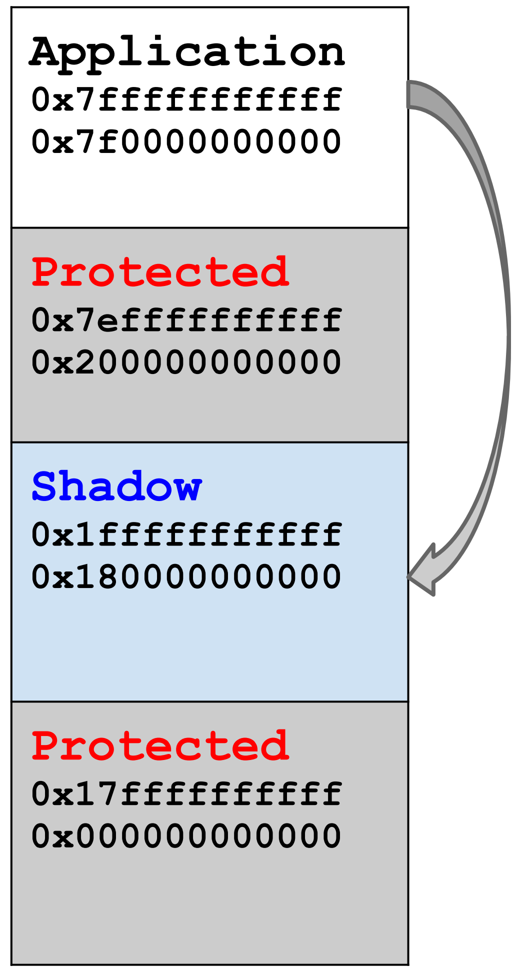
Compiler instrumentation

```
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 shadow mapping (64-bit Linux)

`Shadow = 4 * (Addr & kMask);`

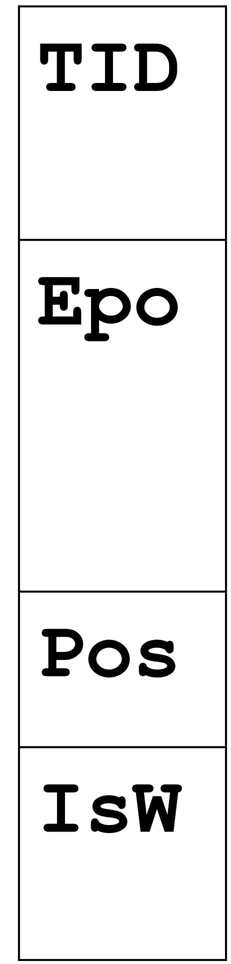


Shadow cell

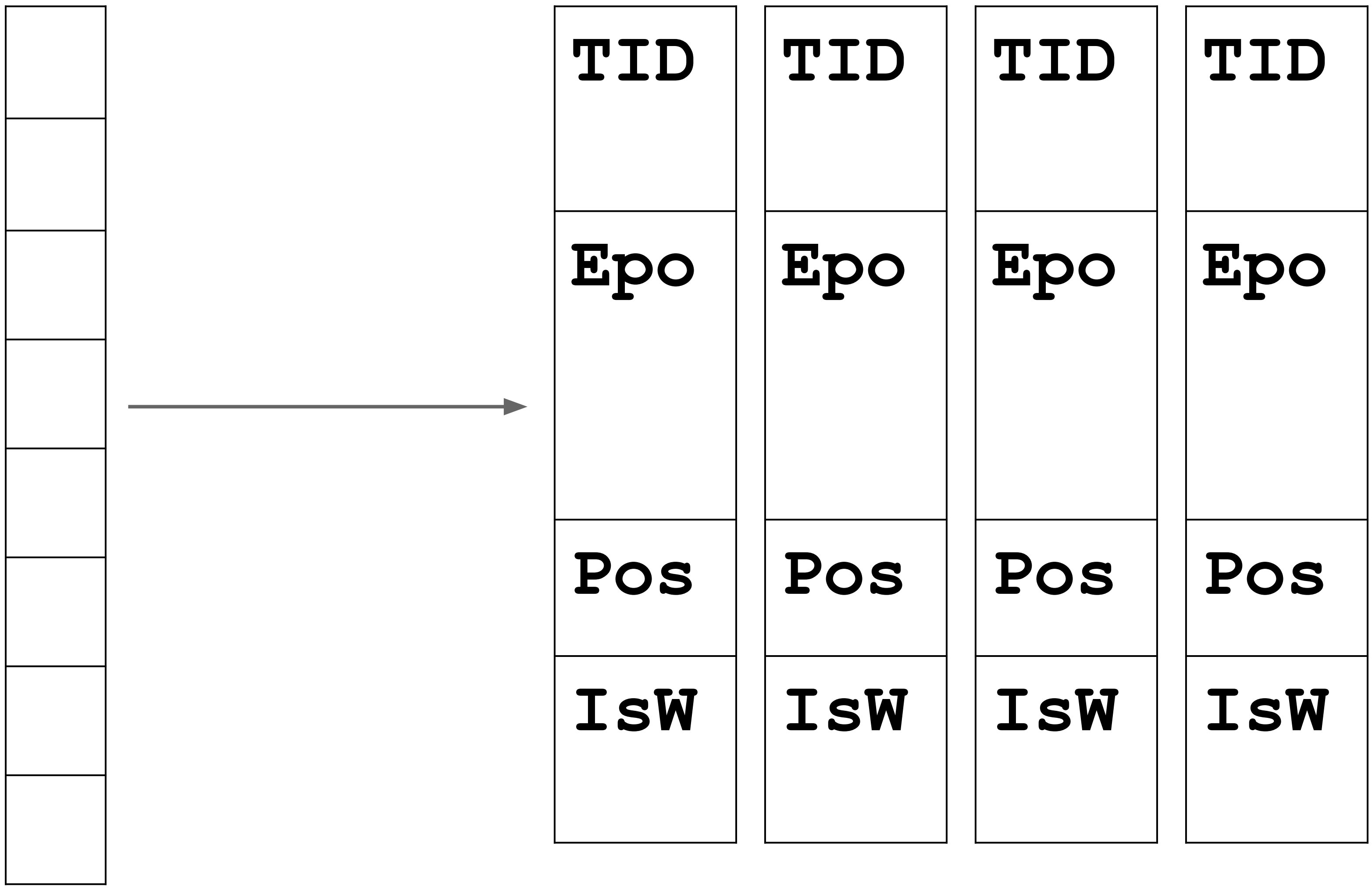
An 8-byte shadow cell represents one memory access:

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

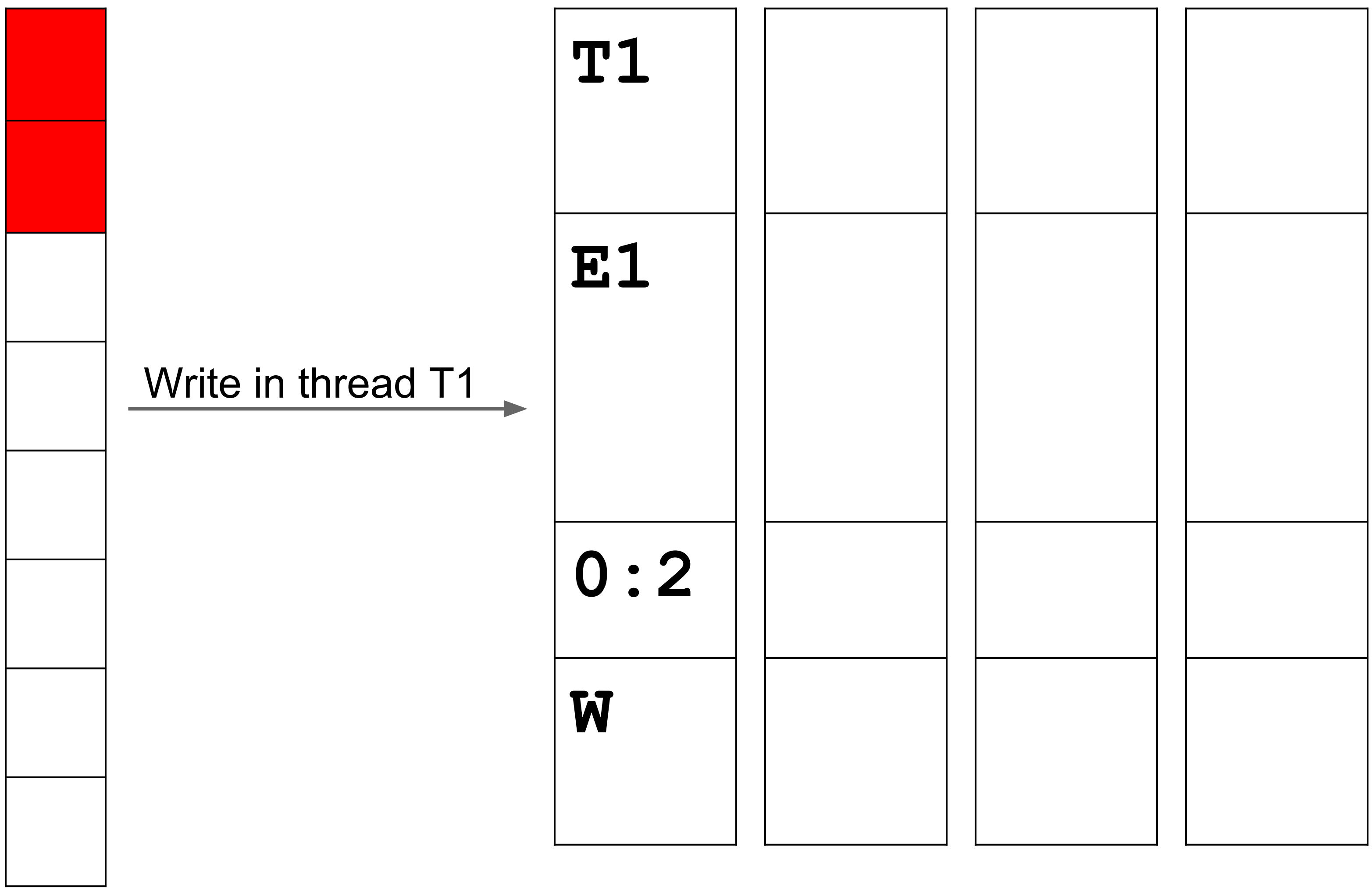
Full information (no more dereferences)



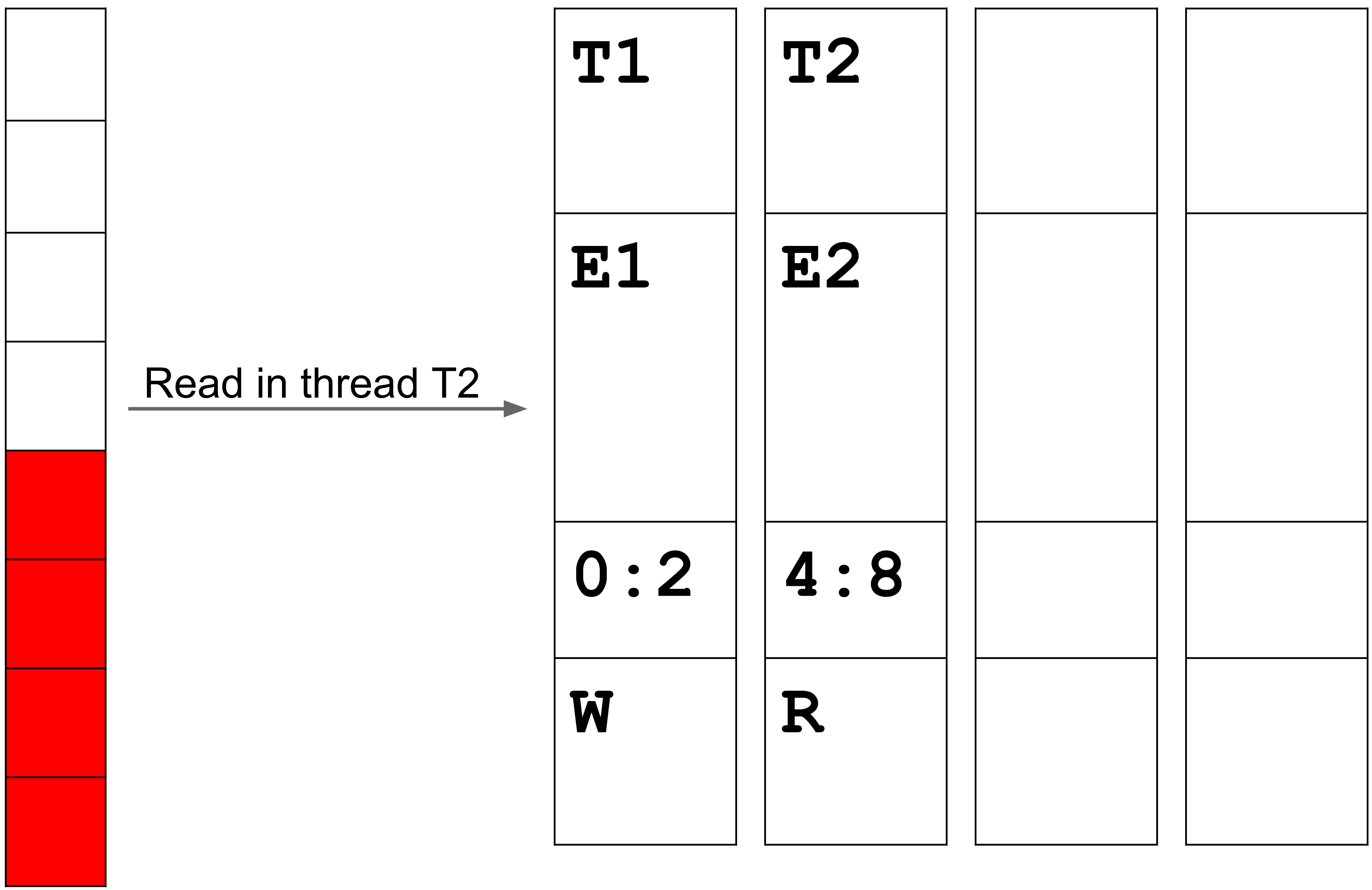
4 shadow cells per 8 app. bytes



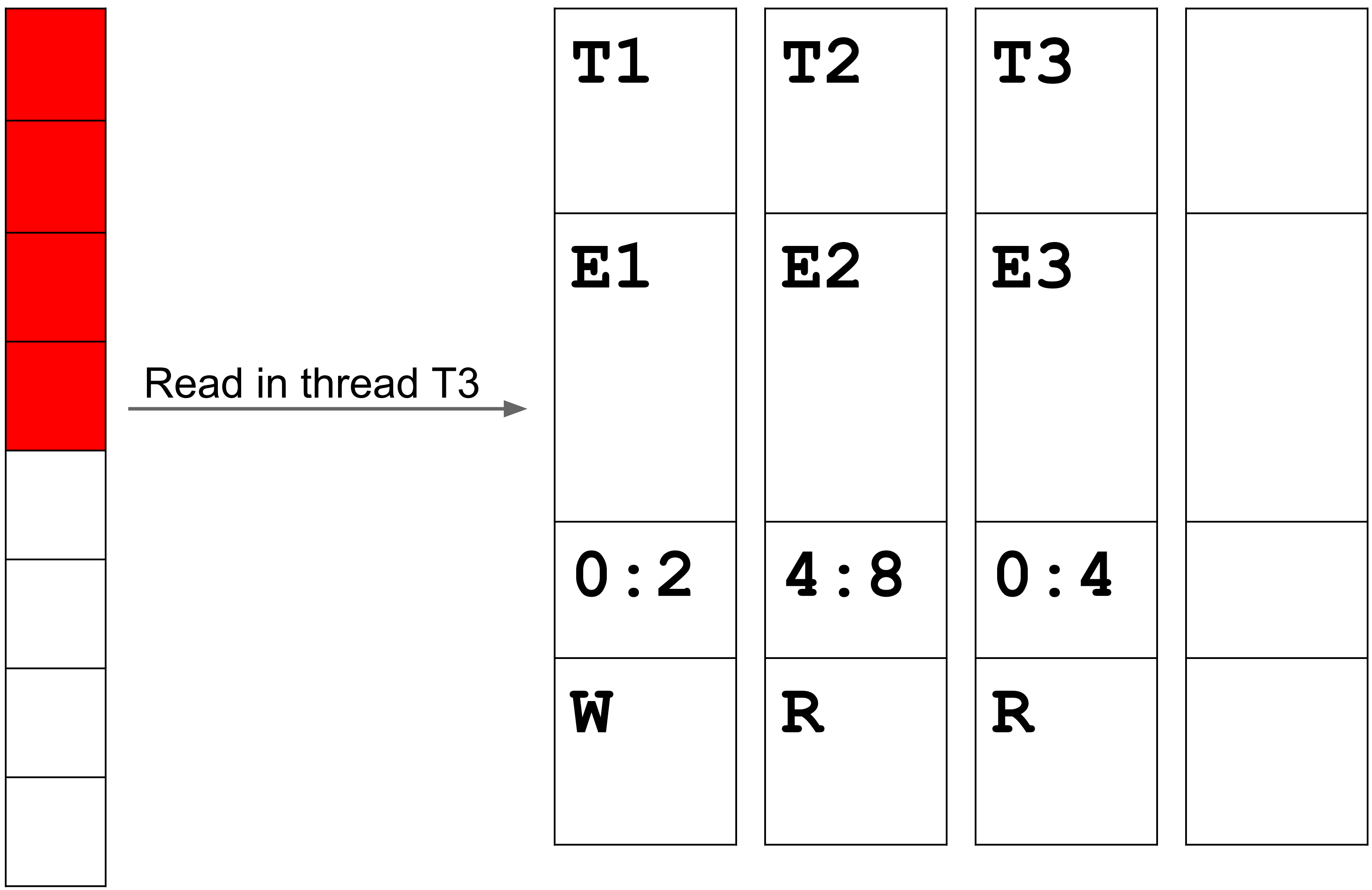
Example: first access



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 cells are filled, one random cell is replaced

Informative reports

- 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
  - Buffer size is configurable
- Replay the event buffer on report
  - Unlimited number of frames

Function interceptors

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

Atomics

- LLVM atomic instructions are replaced with __tsan_* callbacks

```
%0 = load atomic i8* %a acquire, align 1
```

⇓

```
%0 = call i8
@__tsan_atomic8_load(i8* %a, i32 504)
```

TSan slowdown vs clang -O1

Application	TSan1	TSan2	TSan1/TSan2
RPC benchmark	40x	7x	5.5x
Web server test	25x	2.5x	10x
String util test (1 thread)	50x	6x	8.5x

Trophies

- 200+ races in Google server-side apps (C++)
- 80+ races in Go programs
  - 25+ bugs in Go stdlib
- Several races in OpenSSL
  - 1 fixed, ~5 'benign'
- More to come
  - We've just started testing Chrome :)

Key advantages

- Speed
  - > 10x faster than other tools
- Native support for atomics
  - Hard or impossible to implement with binary translation (Helgrind, Intel Inspector)

Limitations

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

MemorySanitizer

uninitialized memory reads (UMR)

MSan report example: UMR

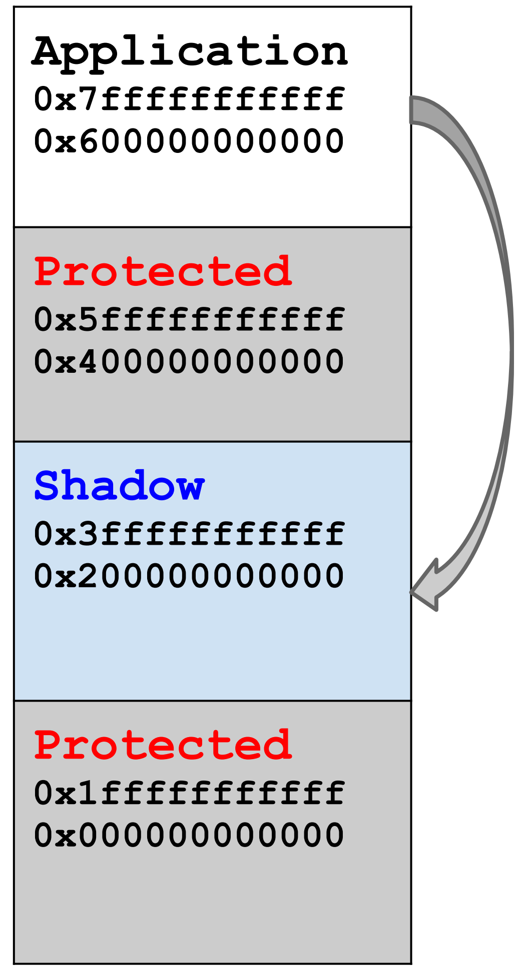
```
int main(int argc, char **argv) {
  int x[10];
  x[0] = 1;
  if (x[argc]) return 1;
  ...
% clang -fsanitize=memory -fPIE -pie  a.c -g
% ./a.out
WARNING: MemorySanitizer: UMR (uninitialized-memory-read)
    #0 0x7ff6b05d9ca7 in main stack_umr.c:4
  ORIGIN: stack allocation: x@main
```

Shadow memory

- Bit to bit shadow mapping
  - 1 means 'poisoned' (uninitialized)
- Uninitialized memory:
  - Returned by malloc
  - Local stack objects (poisoned at function entry)
- Shadow is propagated through arithmetic operations and memory writes
- Shadow is unpoisoned when constants are stored

Direct 1:1 shadow mapping

`Shadow = Addr - 0x400000000000;`



Shadow propagation

- Reporting UMR on first read causes false positives
  - E.g. copying **`struct {char x; int y;}`**
- Report UMR only on some uses (branch, syscall, etc)
  - That's what Valgrind does
- Propagate shadow values through expressions
  - A = B + C: A' = B' | C'
  - A = B & C: A' = (B' & C') | (~B & C') | (B' & ~C)
  - Approximation to minimize false positives/negatives
  - Similar to Valgrind
- Function parameter/retval: shadow is stored in TLS
  - Valgrind shadows registers/stack instead

Tracking origins

- Where was the poisoned memory allocated?

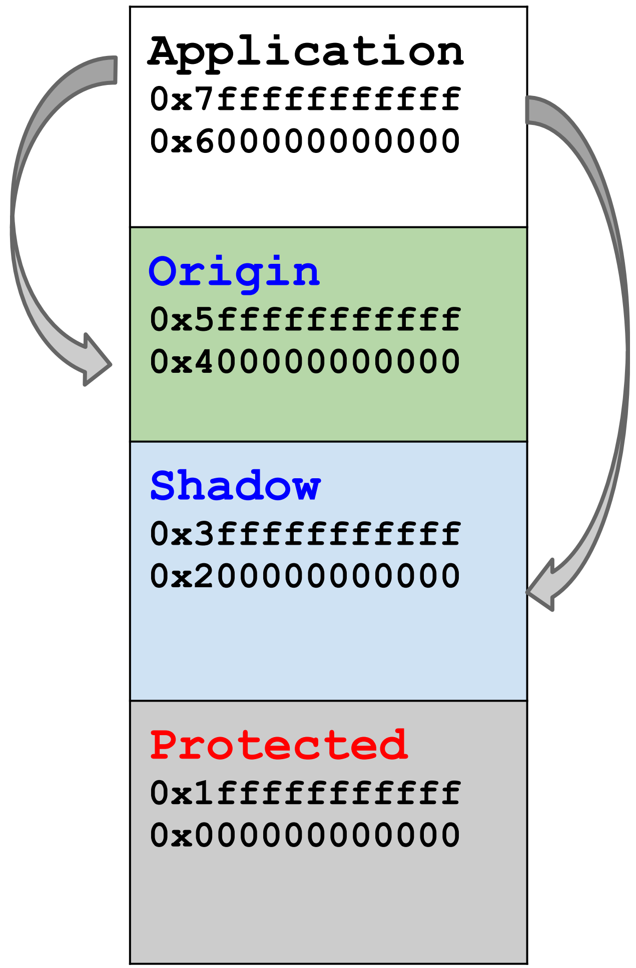
  ```
  a = malloc() ...
  b = malloc() ...
  c = *a + *b  ...
  if (c) ...   // UMR. Is 'a' guilty or 'b'?
  ```

- Valgrind `--track-origins`: propagate the origin of the poisoned memory alongside the shadow

- MemorySanitizer: secondary shadow
  - Origin-ID is 4 bytes, 1:1 mapping
  - 2x additional slowdown

Secondary shadow (origin)

```
Origin = Addr - 0x200000000000;
```



MSan overhead

- Without origins:
  - CPU: 3x
  - RAM: 2x
- With origins:
  - CPU: 6x
  - RAM: 3x + malloc stack traces

Tricky part :(

- Missing any write instruction causes false reports
- Must monitor ALL stores in the program
  - libc, libstdc++, syscalls, etc

Solutions:

- Instrumented libc++, wrappers for libc
  - Works for many "console" apps, e.g. LLVM
- Instrument libraries at run-time
  - DynamoRIO-based prototype (SLOW)
- Instrument libraries statically (is it possible?)
- Compile everything, wrap syscalls
  - Will help AddressSanitizer/ThreadSanitizer too

MSan trophies

- Proprietary console app, 1.3 MLOC in C++
  - Not tested with Valgrind previously
  - 20+ unique bugs in < 2 hours
  - Valgrind finds the same bugs in 24+ hours
  - MSan gives better reports for stack memory
- 1 Bug in LLVM
  - LLVM bootstraps, ready to set regular runs
- A few bugs in Chrome (just started)
  - Have to use DynamoRIO module (MSanDR)
  - 7x faster than Valgrind

Summary (all 3 tools)

- AddressSanitizer (memory corruption)
  - A "must use" for everyone (C++)
  - Supported on Linux, OSX, CrOS, Android,
  - WIP: iOS, Windows, *BSD (?)
- ThreadSanitizer (races)
  - A "must use" if you have threads (C++, Go)
  - Only x86_64 Linux
- MemorySanitizer (uses of uninitialized data)
  - WIP, usable for "console" apps (C++)
  - Only x86_64 Linux

Q&A

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

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

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

ASan/MSan vs Valgrind (Memcheck)

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

Why not a single tool?

- Slowdowns will add up
  - Bad for interactive or network apps
- Memory overheads will multiply
  - ASan redzone vs TSan/MSan large shadow
- Not trivial to implement