Sanitize your C++

Kostya Serebryany, Google
September 11, 2014 @cppcon

C++: shoot yourself in the ~~foot~~ feet

- Buffer overflow (heap, stack, global)
- Heap-use-after-free, stack-use-after-return
- Data race, deadlock
- Use of uninitialized memory
- Memory leak
- Integer overflow
- ...

Why do you care?

- Hard to reproduce and debug bugs
- Sporadic crashes or data corruption
- Excessive resource consumption
- Blah-blah

SECURITY

Do you have enough feet to use C++?

Bullet proof boots for C++:

- AddressSanitizer, aka ASan
  - detects use-after-free and buffer overflows
- ThreadSanitizer, aka TSan
  - detects data races and deadlocks
- MemorySanitizer, aka MSan
  - detects uninitialized memory reads
- UndefinedBehaviorSanitizer, aka UBSan
  - detects “simple” undefined behaviors

AddressSanitizer

addressability bugs

AddressSanitizer overview

- Finds
  - buffer overflows (stack, heap, globals)
  - heap-use-after-free, stack-use-after-return
  - leaks, ODR violations, init-order fiasco, double-free, etc
- Compiler module (LLVM, GCC)
  - instruments all loads/stores
  - inserts redzones around stack and global Variables
- Run-time library
  - malloc replacement (redzones, quarantine)
  - Bookkeeping for error messages

ASan report example: global-buffer-overflow

```
int global_array[100] = {-1};
int main(int argc, char **argv) {
  return global_array[argc + 100];  // BOOM
}
% clang++ -O1 -fsanitize=address a.cc ; ./a.out

==10538== ERROR: AddressSanitizer global-buffer-overflow
READ of size 4 at 0x000000415354 thread T0
    #0 0x402481 in main a.cc:3
    #1 0x7f0a1c295c4d in __libc_start_main ??:0
    #2 0x402379 in _start ??:0
0x000000415354 is located 4 bytes to the right of global
  variable 'global_array' (0x4151c0) of size 400
```

ASan report example: stack-buffer-overflow

```
int main(int argc, char **argv) {
  int stack_array[100];
  stack_array[1] = 0;
  return stack_array[argc + 100];  // BOOM
}
% clang++ -O1 -fsanitize=address a.cc; ./a.out

==10589== ERROR: AddressSanitizer stack-buffer-overflow
READ of size 4 at 0x7f5620d981b4 thread T0
    #0 0x4024e8 in main a.cc:4
Address 0x7f5620d981b4 is located at offset 436 in frame
  <main> of T0's stack:
  This frame has 1 object(s):
    [32, 432) 'stack_array'
```

ASan report example: heap-buffer-overflow

```
int main(int argc, char **argv) {
  int *array = new int[100];
  int res = array[argc + 100];  // BOOM
  delete [] array;
  return res;
}
% clang++ -O1 -fsanitize=address a.cc; ./a.out

==10565== ERROR: AddressSanitizer heap-buffer-overflow
READ of size 4 at 0x7fe4b0c76214 thread T0
    #0 0x40246f in main a.cc:3
0x7fe4b0c76214 is located 4 bytes to the right of 400-
  byte region [0x7fe..., 0x7fe...)
allocated by thread T0 here:
    #0 0x402c36 in operator new[](unsigned long)
    #1 0x402422 in main a.cc:2
```

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 report example: container-overflow

```
#include <vector>
int main() {
  std::vector<int> V(8);
  V.resize(5);
  return V.data()[6]; // Between V.size() and V.capacity()
}
% clang++ -O1 -fsanitize=address a.cc && ./a.out
==4729==ERROR: AddressSanitizer: container-overflow
READ of size 4 at 0x60300000eff8 thread T0
    #0 0x486866 in main a.cc:5

0x6...f8 is located 24 bytes inside of 32-byte region
allocated by thread T0 here:
    #0 0x46e1e1 in operator new(unsigned long) ...
    #6 0x486730 in main a.cc:3
```

ASan report example: stack-use-after-return

```
int *g;
void LeakLocal() {
  int local;
  g = &local;
}
```

```
int main() {
  LeakLocal();
  return *g;
}
```

```
% clang -g -fsanitize=address a.cc
% ASAN_OPTIONS=detect_stack_use_after_return=1 ./a.out
```

```
==19177==ERROR: AddressSanitizer: stack-use-after-return
READ of size 4 at 0x7f473d0000a0 thread T0
    #0 0x461ccf in main   a.cc:8

Address is located in stack of thread T0 at offset 32 in frame
    #0 0x461a5f in LeakLocal()  a.cc:2
  This frame has 1 object(s):
    [32, 36) 'local' <== Memory access at offset 32
```

ASan report example: init-order-fiasco

```
// i1.cc
extern int B;
int A = B;
int main() {
  return A;
}
```

```
// i2.cc
#include <stdlib.h>
int B = atoi("123");
```

```
% clang -g -fsanitize=address i1.cc i2.cc; ./a.out

==19504==ERROR: AddressSanitizer: initialization-order-fiasco
READ of size 4 at 0x000001aaff60 thread T0
    #0 0x414fa3 in __cxx_global_var_init  i1.cc:2
    #1 0x415015 in global constructors keyed to a  i1.cc:5

0x000001aaff60 is located 0 bytes inside
  of global variable 'B' from 'i2.cc' (0x1aaff60) of size 4
```

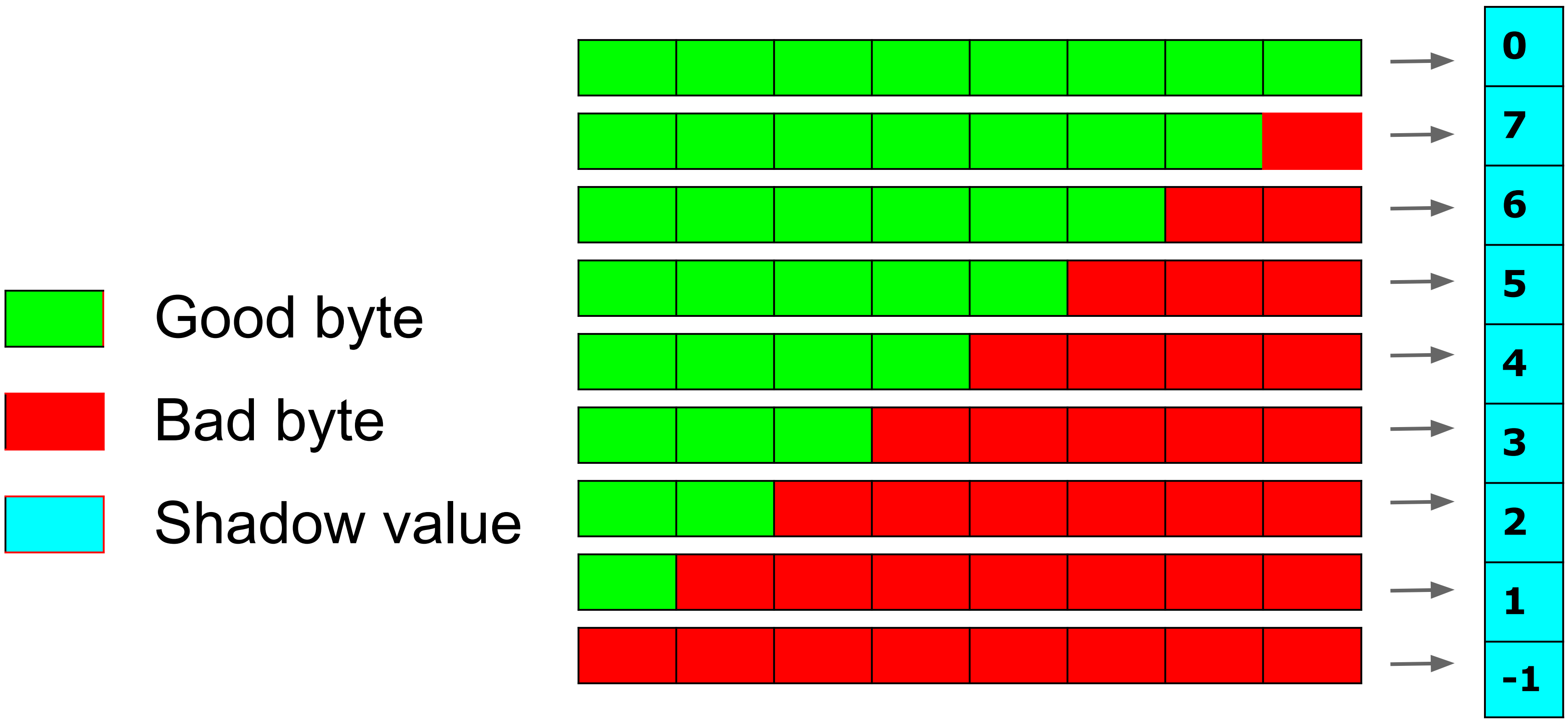
ASan report example: memory leak

```
int *g = new int;
int main() {
  g = 0; // Lost the pointer.
}

% clang -g -fsanitize=address a.cc; ./a.out

==19894==ERROR: AddressSanitizer: detected memory leaks

Direct leak of 4 byte(s) in 1 object(s) allocated from:
    #0 0x44a3b1 in operator new(unsigned long)
    #1 0x414f66 in __cxx_global_var_init  a.cc:1
```

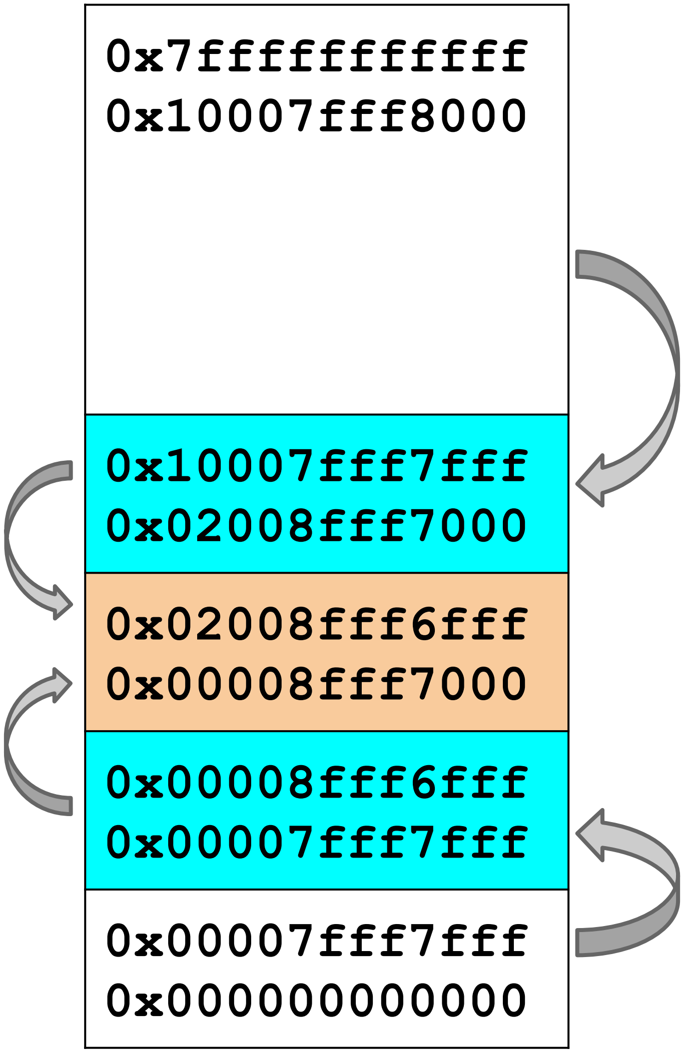
ASan shadow byte

Any aligned 8 bytes may have 9 states:

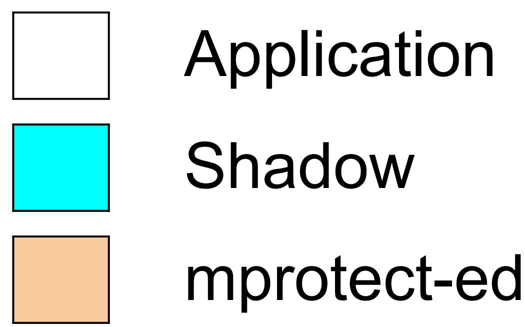
N good bytes and 8 - N bad (0<=N<=8)



ASan virtual address space




```
Shadow =
 Addr / 8 + kOffset
```



ASan instrumentation: 8-byte access

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

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

ASan instrumentation: N-byte access (1, 2, 4)

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

↓

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

Instrumentation example (x86_64)

```
mov     %rdi,%rax
shr     $0x3,%rax               # shift by 3
cmpb    $0x0,0x7fff8000(%rax) # load shadow
je 1f   <foo+0x1f>
ud2a                            # generate SIGILL*
movq    $0x1234,(%rdi)          # original store
```

*** May use call instead of UD2**

Instrumenting stack frames

```
void foo() {

  char a[328];




  <------------- CODE ------------->

}
```

Instrumenting stack frames

```
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
```

Malloc replacement

- Insert redzones around every allocation
  - poison redzones on malloc
- Delay the reuse of freed memory
  - poison the entire memory region on free
- Collect stack traces for every malloc/free

ASan *marketing* slide

- 2x slowdown (Valgrind: 20x and more)
- 1.5x-3x memory overhead

- 3000+ bugs found in Chrome in 3 years

- 3000+ bugs found in Google server software

- 2000+ bugs everywhere else
    - Firefox, FreeType, FFmpeg, WebRTC, libjpeg-turbo, Perl, Vim, LLVM, GCC, MySQL

ASan and Chrome

- Chrome was the first ASan user (May 2011)
- Now all existing tests are running with ASan
- Fuzzing at massive scale (ClusterFuzz), 2000+ cores
  - Generate test cases, minimize, de-duplicate
  - Find regression ranges, verify fixes
- Over 3000 security bugs found in 3 years
  - External researchers found 100+ bugs
- Similar situation with Mozilla Firefox

ThreadSanitizer

concurrency bugs

ThreadSanitizer

- Detects data races and deadlocks
- Compile-time instrumentation (LLVM, GCC)
  - Intercepts all reads/writes
- Run-time library
  - Malloc replacement
  - Intercepts all synchronization
  - Handles reads/writes

TSan report example: data race

```
int X;
std::thread t([&]{X = 42;});
X = 43;
t.join();
```

```
% clang -fsanitize=thread -g race.cc && ./a.out
WARNING: ThreadSanitizer: data race (pid=25493)
  Write of size 4 at 0x7fff7f10e338 by thread T1:
    #0 main::$_0::operator()() const race.cc:4
    ...
  Previous write of size 4 at 0x7...8 by main thread:
    #0 main race.cc:5

  Location is stack of main thread.
```

TSan report example: deadlock

```
// mu0 => mu1
lock_guard<mutex> l0(mu0);
lock_guard<mutex> l1(mu1);

…

// mu1 => mu0
lock_guard<mutex> l1(mu1);
lock_guard<mutex> l0(mu0);
...
```

```
WARNING: ThreadSanitizer: lock-order-inversion (potential deadlock)
Cycle in lock order graph: M0 => M1 => M0

M1 acquired here while holding mutex M0:
    #1 main mutex_cycle2.c:10

M0 previously acquired by the same thread here:
    #1 main mutex_cycle2.c:9

M0 acquired here while holding mutex M1:
    #1 main mutex_cycle2.c:16

M1 previously acquired by the same thread here:
    #1 main mutex_cycle2.c:15
```

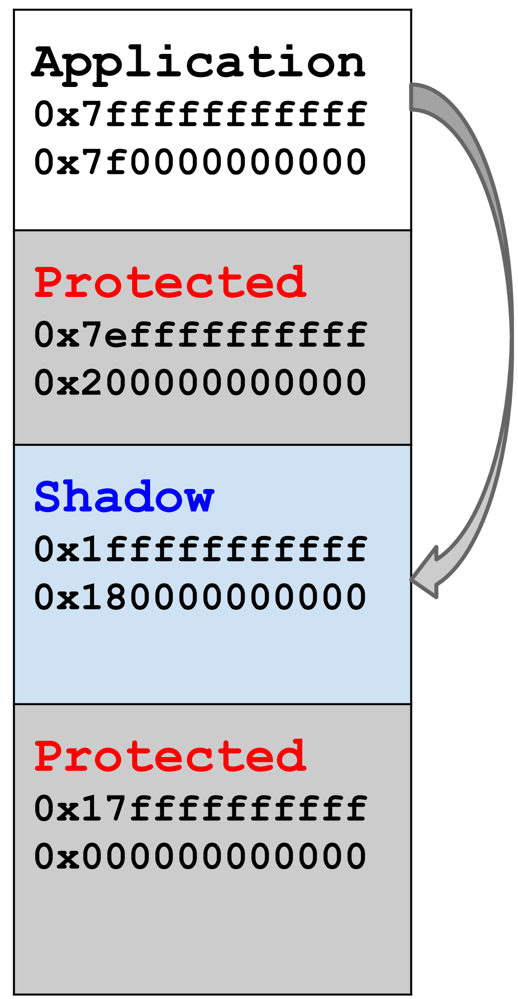
Compiler instrumentation

```
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);
```

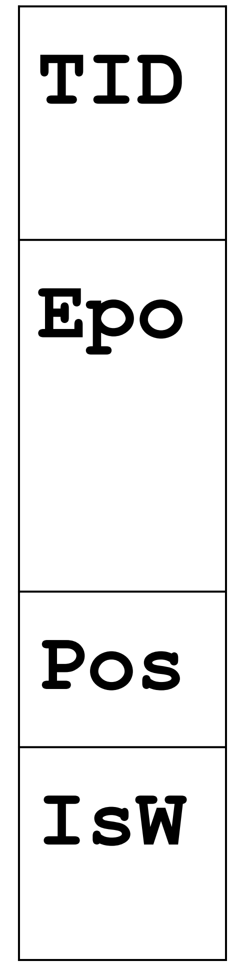


Shadow cell

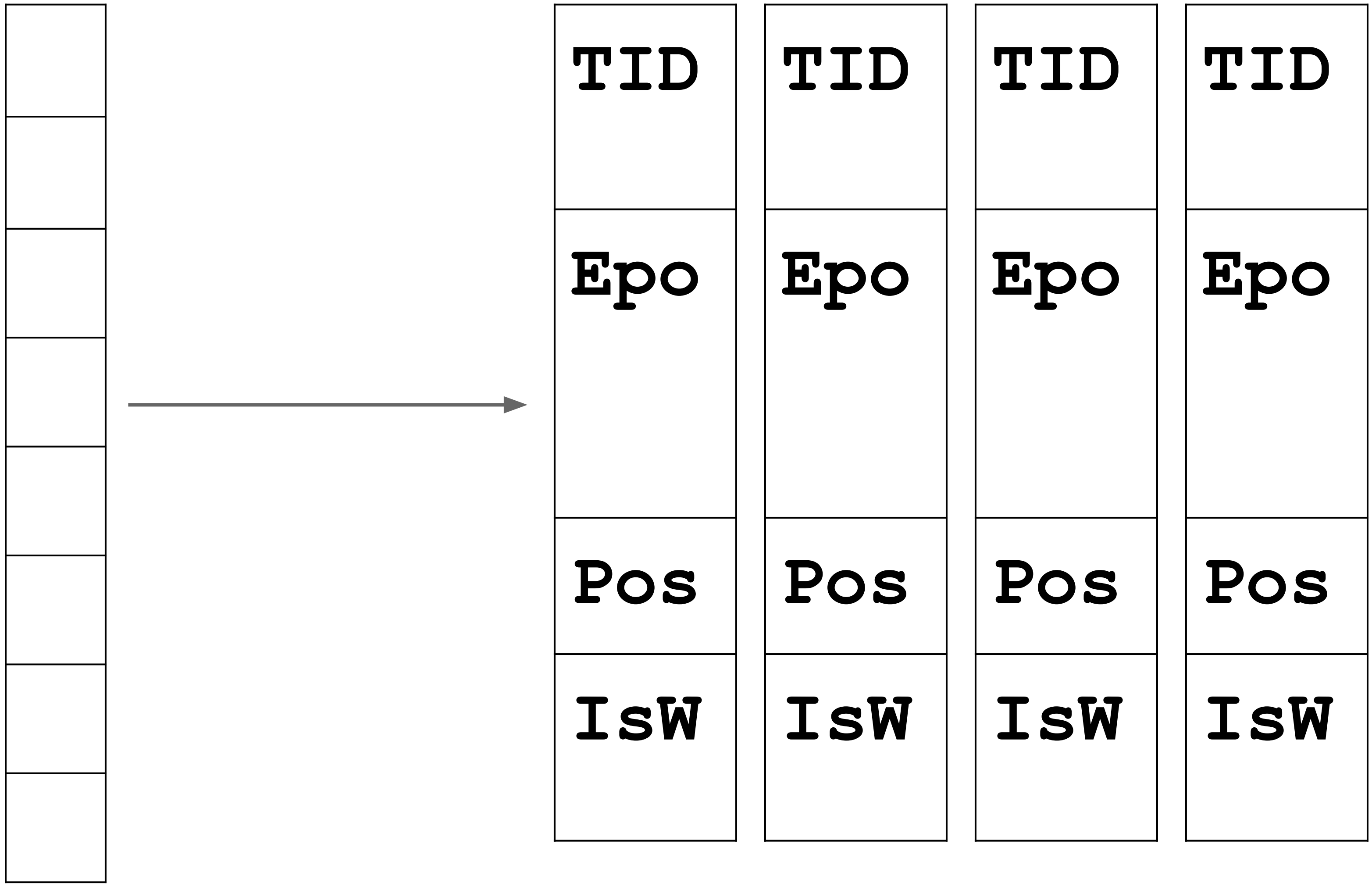
An 8-byte shadow cell represents one memory access:

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

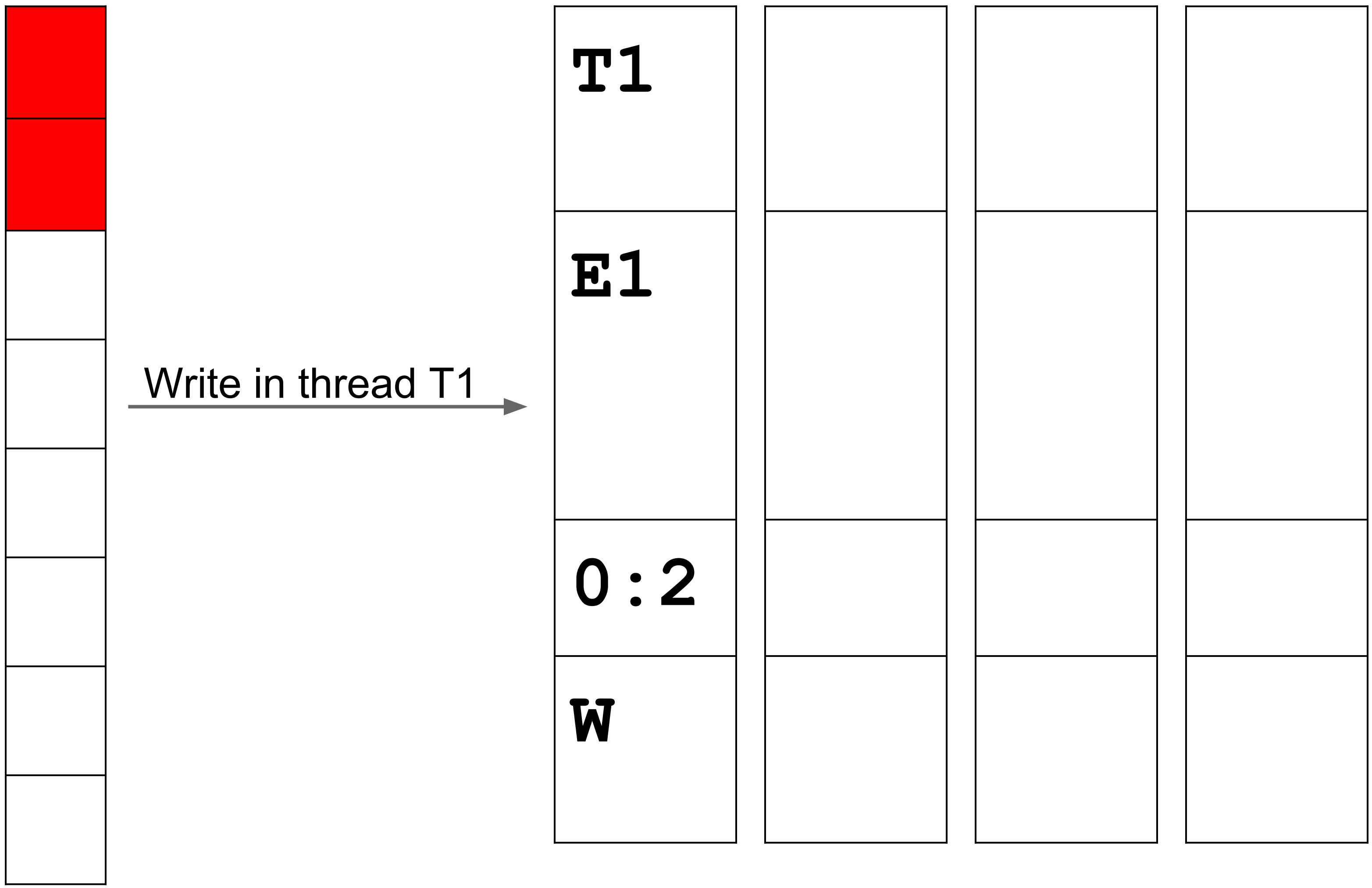
Full information (no more dereferences)



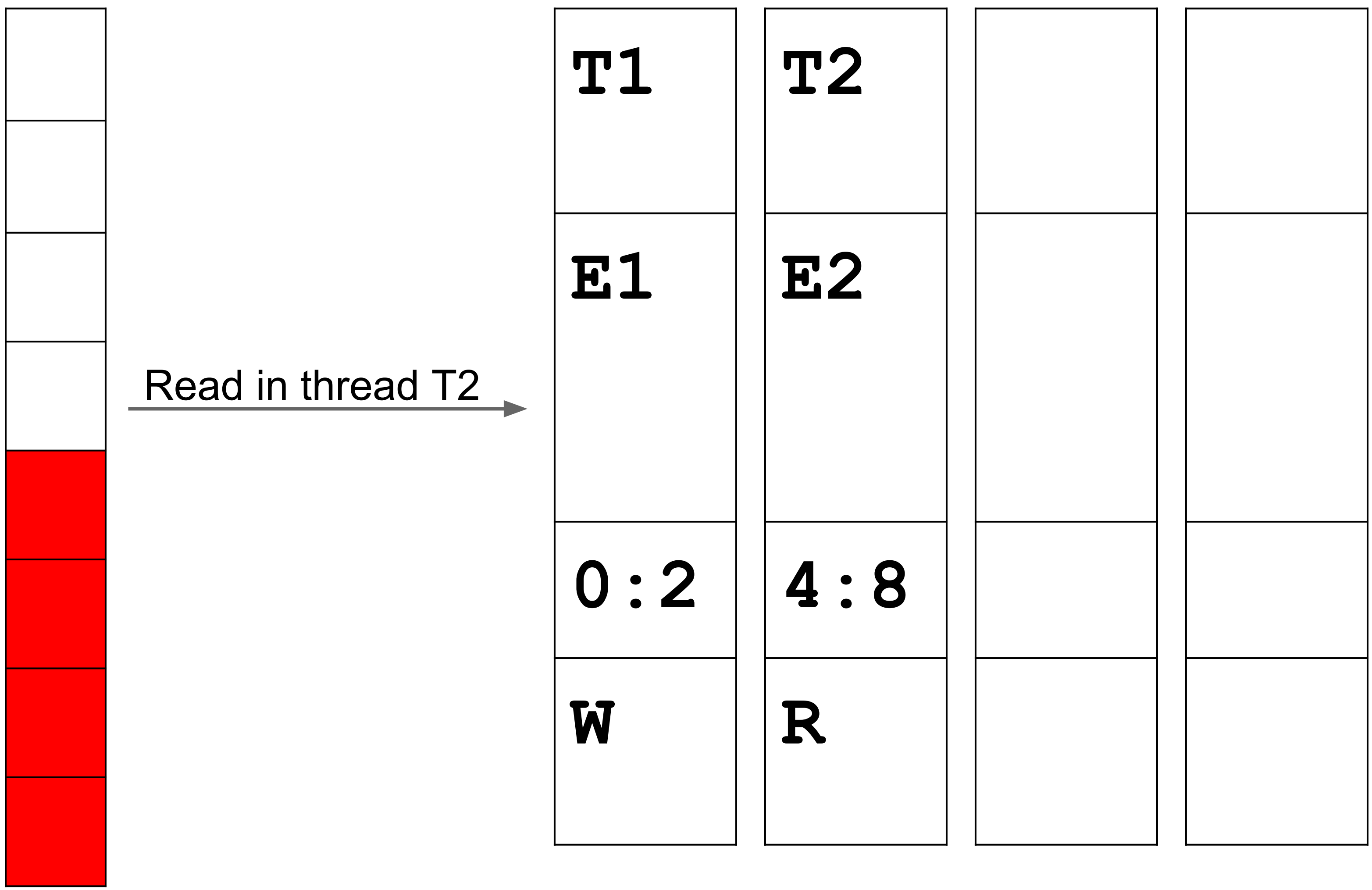
4 shadow cells per 8 app. bytes



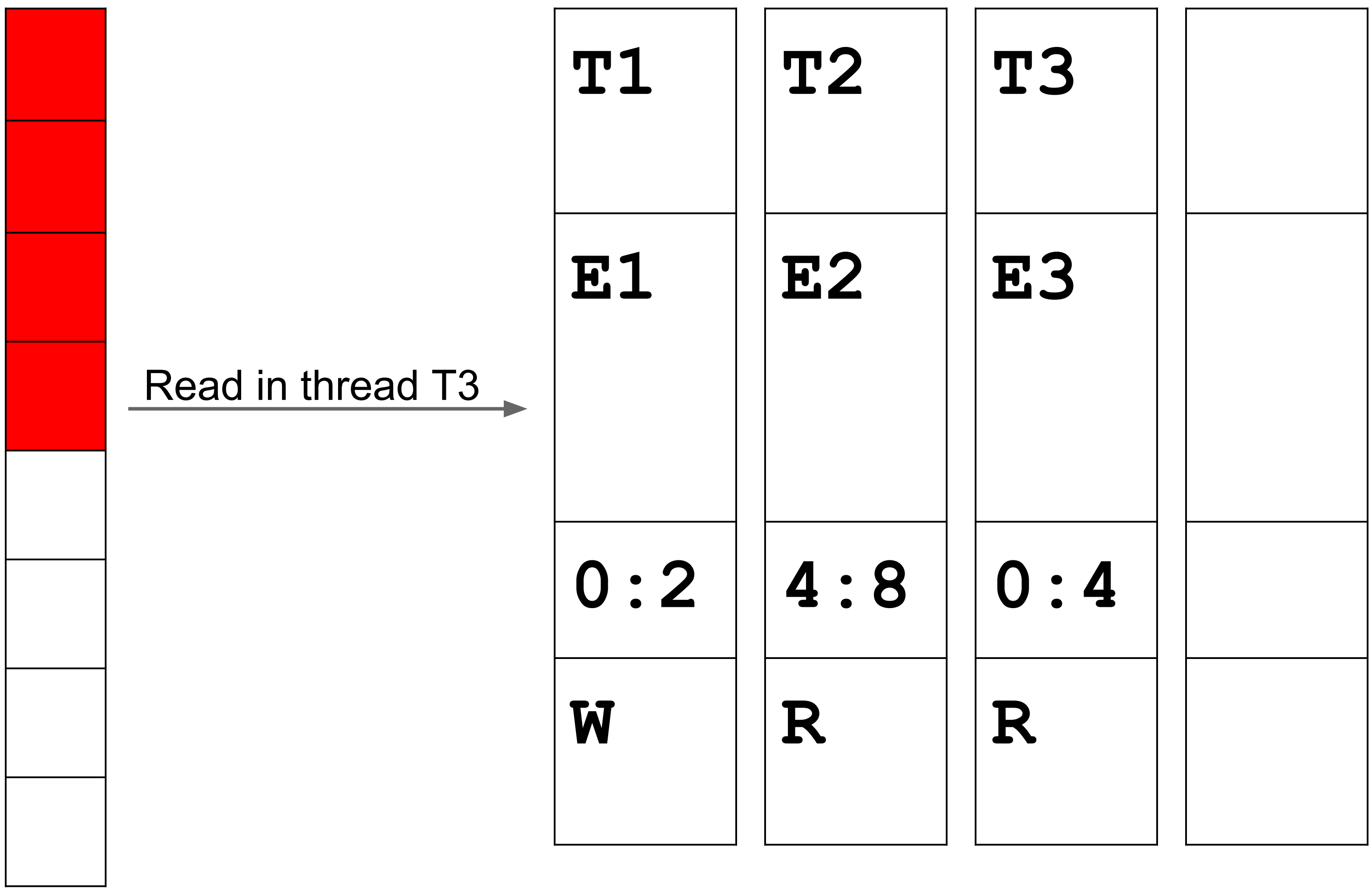
Example: first access



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
- Somewhat similar to FastTrack (PLDI'09)

Stack trace for previous access

- Important to understand the report

- 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

TSan overhead

- CPU: 4x-10x
- RAM: 5x-8x

Trophies

- 500+ races in C++ Google server-side apps
  - Scales to huge apps
- 100+ races in Go programs
  - 25+ bugs in Go stdlib
- 100+ races in Chromium

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
  - Relies on atomic 64-bit load/store
  - Requires lots of RAM
- Does not instrument (yet):
  - pre-built libraries
  - inline assembly

MemorySanitizer

uninitialized memory reads (UMR)

MSan report example

```
int main(int argc, char **argv) {
  int x[10];
  x[0] = 1;
  return x[argc];
}
% clang -fsanitize=memory a.c -g; ./a.out

WARNING: Use of uninitialized value
    #0 0x7f1c31f16d10 in main a.cc:4
Uninitialized value was created by an
allocation of 'x' in the stack frame of
function '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 unpoisoned when constants are stored

Shadow propagation

Reporting every load of uninitialized data is too noisy.

```
struct {
  char x;
  // 3-byte padding
  int y;
}
```

It's OK to copy uninitialized data around.

Uninit calculations are OK, too, as long as the result is discarded. People do it.

Shadow propagation

$A = B << C$: $A' = B' << C$

$A = B \& C$: $A' = (B' \& C') \;|\; (B \& C') \;|\; (B' \& C)$

$A = B + C$: $A' = B' \,|\, C'$ *(approx.)*

Report errors only on some uses: conditional branch, syscall argument (visible side-effect).

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

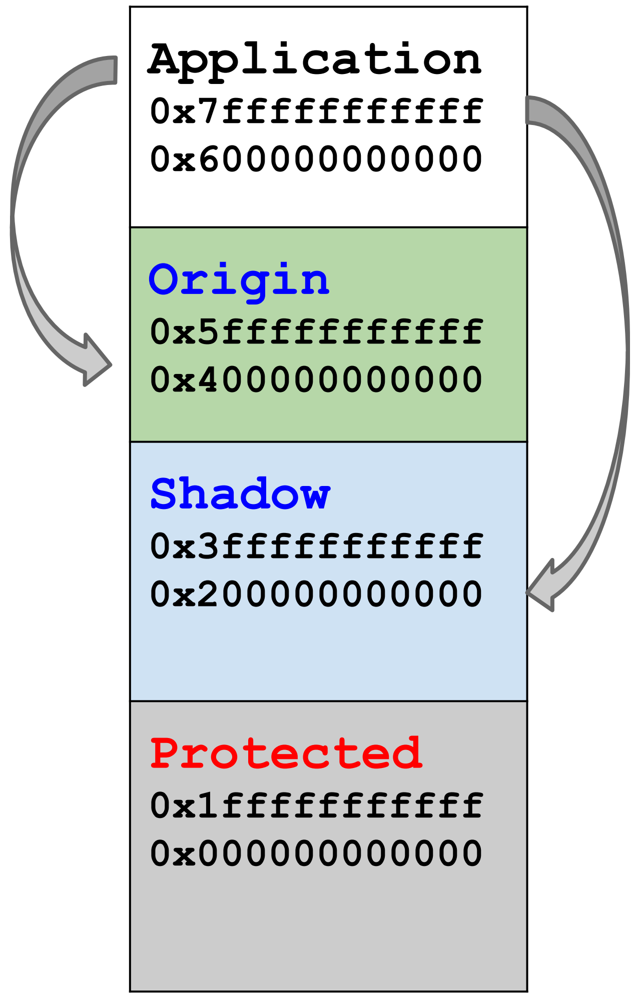
Advanced origin tracking

```
int arr[2];
void shift() {arr[1] = arr[0];}
void push(int *p) {
  shift();
  arr[0] = *p;
}
int pop() {
  int x = arr[1];
  shift();
  return x;
}
void func1() {
  int local_var;  // OUCH
  push(&local_var);
}
int main() {
  func1();
  shift();
  return pop();
}
```

```
MemorySanitizer: use-of-uninitialized-value
  #0 0x7f60954bdaf7 in main test.cc:19

Uninitialized value was stored to memory at
  #0 0x7f60954bd73f in pop test.cc:8
  #1 0x7f60954bdaaf in main test.cc:19

Uninitialized value was stored to memory at
  #0 0x7f60954bd3e3 in shift test.cc:2
  #1 0x7f60954bda95 in main test.cc:18

Uninitialized value was stored to memory at
  #0 0x7f60954bd5f6 in push test.cc:5
  #1 0x7f60954bd7ef in func1 test.cc:14
  #2 0x7f60954bda90 in main est.cc:17

Uninitialized value was created by an
allocation of 'local_var' in the stack frame of
function 'func1'
  #0 0x7f60954bd790 in func1 test.cc:12
```

Shadow mapping

```
Shadow = Addr - 0x400000000000;
Origin = Addr - 0x200000000000;
```



MSan overhead

- Without origins:
  - CPU: 2.5x
  - RAM: 2x
- With origins:
  - CPU: 5x
  - RAM: 3x

Tricky part :(

Missing any write causes false reports.

- Libc
  - Solution: function wrappers

- Inline assembly
  - Openssl, libjpeg_turbo, etc

- JITs (e.g. V8)

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 esp. for stack memory
- 20+ in LLVM
  - Regressions caught by regular LLVM bootstrap
- 400+ bugs in Google server-side code
- 200+ bugs in Chromium

UndefinedBehaviorSanitizer

Various “simple” bugs

UBSan report example: int overflow

```
int main(int argc, char **argv) {
  int t = argc << 16;
  return t * t;
}
```

```
% clang -fsanitize=undefined a.cc -g; ./a.out
```

```
a.cc:3:12: runtime error:
signed integer overflow: 65536 * 65536
cannot be represented in type 'int'
```

UBSan report example: invalid shift

```
int main(int argc, char **argv) {
  return (1 << (32 * argc)) == 0;
}
```

```
% clang -fsanitize=undefined a.cc -g; ./a.out
```

```
a.cc:2:13: runtime error: shift exponent 32 is
too large for 32-bit type 'int'
```

UBSan deployment

- Main challenge: too many real bugs found
- May use only a subset of checks:

-fsanitize=alignment	-fsanitize=returns-nonnull-attribute
-fsanitize=bool	-fsanitize=shift
-fsanitize=bounds	-fsanitize=signed-integer-overflow
-fsanitize=enum	-fsanitize=unreachable
-fsanitize=float-cast-overflow	-fsanitize=unsigned-integer-overflow
-fsanitize=float-divide-by-zero	-fsanitize=vla-bound
-fsanitize=function	-fsanitize=vptr
-fsanitize=integer-divide-by-zero	-fsanitize=object-size
-fsanitize=null	-fsanitize=return

- Slowdown varies between 0% and 50%

Wrapping up...

Current status of Sanitizers

- ASan
  - Clang 3.1+ and GCC 4.8+
  - i386, x86_64, ARM, AArch64, Power, MIPS, Sparc,...
  - Linux, OSX, Windows, Android, FreeBSD, iOS, …
- TSan:
  - Clang 3.2+ and GCC 4.8+
  - Linux x86_64
- MSan:
  - Clang 3.3+, Linux x86_64
- UBSan:
  - Clang 3.3+ and GCC 4.9 (subset)
  - Linux x86_64, OSX

By the way...

ASan for Linux Kernel

Deployment challenges

- Real bugs that need to be fixed
- “Benign” bugs, especially races
  - even though there is no such thing!

- Memory overhead
  - limited RAM on a device or VM
- CPU overhead is minor issue
  - but it has a cost in $$

- Run sanitizers in production to catch the last 1% of bugs

Plea to compiler vendors

Please, implement AddressSanitizer
and other sanitizers in your C++ compiler!

- ASan compiler module is tiny:
    - Clang: 1.8 KLOC
    - GCC: 2.7KLOC
- ASan run-time library may be reused
    - BSD-like license

Challenge for the Software engineering community

All of the code needs to be available for re-compilation to get maximal possible benefit from (static or dynamic) code analysis tools

Q&A

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

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

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

http://clang.llvm.org/docs/UsersManual.html

Quiz: find all bugs

```
#include <thread>  // C++11
int main() {
  int *a = new int[4];
  int *b = new int[4];
  std::thread t{[&](){b++;}};
  delete a;
  t.detach();
  return *a + (*++b) + b[3];
}
```

Dynamic vs static analysis

Static analysis:

+ Checks all code
+ Does not require tests
- Complex methods don’t scale
- False positives

Dynamic analysis:

- Requires very good test coverage
- Requires to run tests, adds slowdown
+ Finds bugs that static analysis can not find in theory
+ No false positives

ASan/MSan vs Valgrind (Memcheck)

	Valgrind	ASan	MSan
Heap out-of-bounds	YES	YES	
Stack out-of-bounds		YES	
Global out-of-bounds		YES	
Use-after-free	YES	YES	
Use-after-return		YES	
Uninitialized reads	YES		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