AddressSanitizer/ThreadSanitizer for Linux Kernel and userspace.

Konstantin Serebryany, Dmitry Vyukov

Linux Collaboration Summit
15 April 2013

Agenda

- AddressSanitizer, a memory error detector (userspace)
- ThreadSanitizer, a data race detector (userspace)
- Thoughts on AddressSanitizer for Linux Kernel
- Our requests to the Kernel

AddressSanitizer (ASan)

a memory error detector

Memory Bugs in C++

- Buffer overflow
    - Heap
    - Stack
    - Globals
- Use-after-free (dangling pointer)
- Double free
- Invalid free
- Overapping memcpy parameters
- ...

AddressSanitizer overview

- Compile-time instrumentation module
  - Platform independent
- Run-time library
  - Supports Linux, OS X, Android, Windows
- Released in May 2011
- Part of LLVM since November 2011
- Part of GCC since March 2013

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

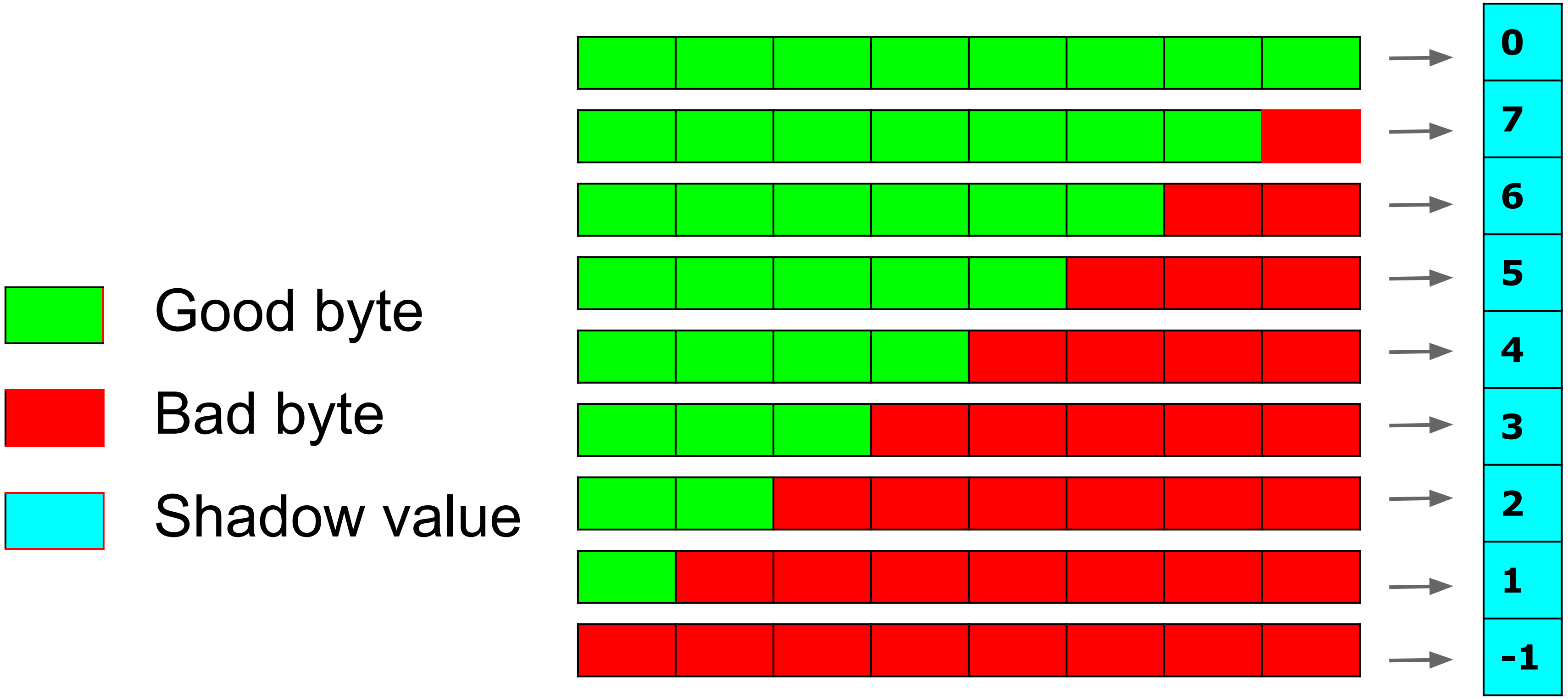
```
int main(int argc, char **argv) {
    int *array = new int[100];
    delete [] array;
    return array[argc];  // BOOM
}
% clang++ -O1 -fsanitize=address a.cc && ./a.out

==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 shadow byte

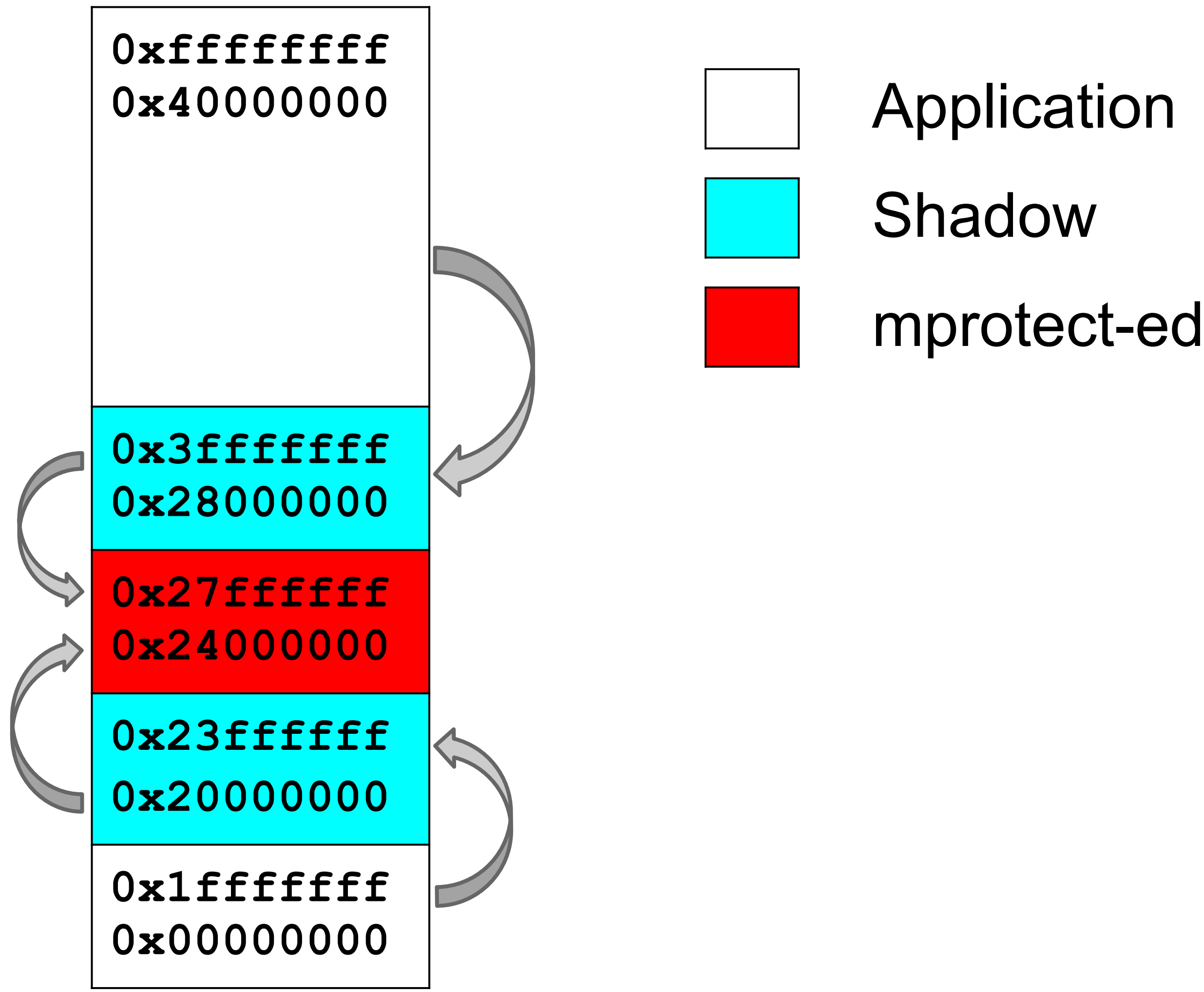
Any aligned 8 bytes may have 9 states:

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

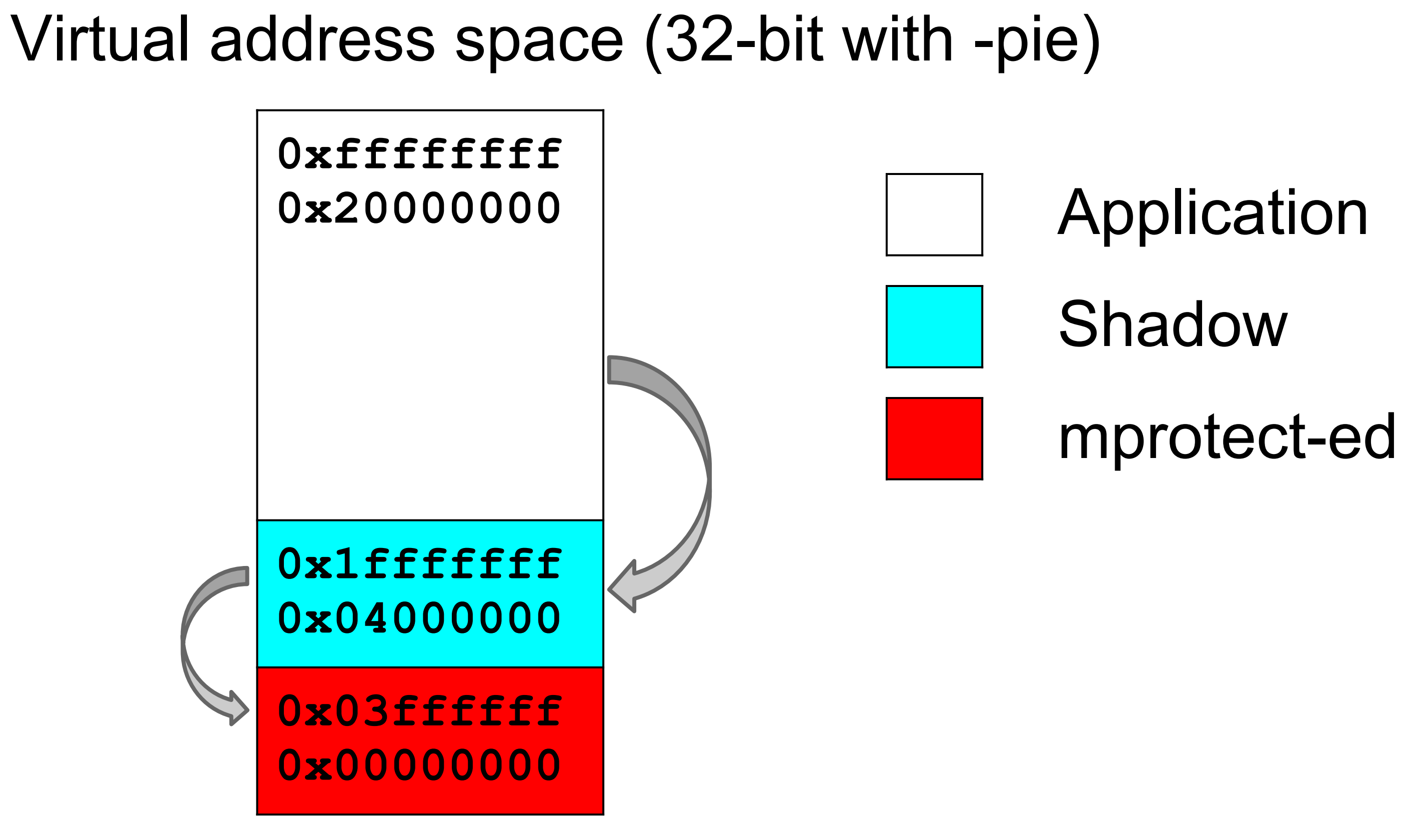


Mapping: `Shadow = (Addr>>3) + Offset`

Virtual address space (32-bit with)



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



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)

```
mov   %rdi,%rax           # address is in %rdi
shr   $0x3,%rax           # shift by 3
cmpb  $0x0,0x7fff8000(%rax) # shadow ? 0
je 1f <foo+0x1f>
callq __asan_report_store8  # Report error
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

- SPEC 2006: average slowdown is < 2x
    - "**`clang -O2`**" vs "**`clang -O2 -fsanitize=address -fno-omit-frame-pointer`**"
- Almost no slowdown for GUI programs (e.g. Chrome)
    - They don't consume all of CPU anyway
- 1.5x - 3x slowdown for server side apps with -O2

Memory overhead

- Heap redzones
  - 16-2048 bytes per allocation, typically 20% of size
- Stack redzones: 32-63 bytes per addr-taken local var
- Global redzones: 32+ bytes per global
- Fixed size Quarantine (256M)
- (Heap + Globals + Stack + Quarantine) / 8 (shadow)

- **Typical overall memory overhead is 2x-3x**

- Stack size increase up to 3x
- `mmap MAP_NORESERVE` 1/8-th of all address space
  - 20T on 64-bit
  - 0.5G on 32-bit

Trophies

- Chromium (including WebKit); in first 10 months
  - heap-use-after-free: 201
  - heap-buffer-overflow: 73
  - global-buffer-overflow: 8
  - stack-buffer-overflow: 7
- Mozilla
- FreeType, FFmepeg, libjpeg-turbo, Perl, Vim, LLVM, GCC, WebRTC, MySQL, ...
- Google server-side apps

Future work

- Avoid redundant checks (static analysis)
- Instrument or recompile libraries
- Instrument inline assembler
- Adapt to use in a kernel
    - discussed later in this talk!

C++ is suddenly
a much safer language

MemorySanitizer (MSan)

finds uses of uninitialized memory

(not in this talk)

ThreadSanitizer (TSan)

a data race detector

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
    #1 main a.c:3
```

ThreadSanitizer v1

- Used since 2009
- Based on Valgrind
- Slow (20x-400x slowdown)
  - Still, found thousands races
  - Also, faster than others
- Other race detectors for C/C++:
  - 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

Execution Slowdown

Application	Tsan1	Tsan2	Tsan1/Tsan2
RPC benchmark	428	2.8	155
Server app test	26	1.8	15
String util test	40	3.4	12

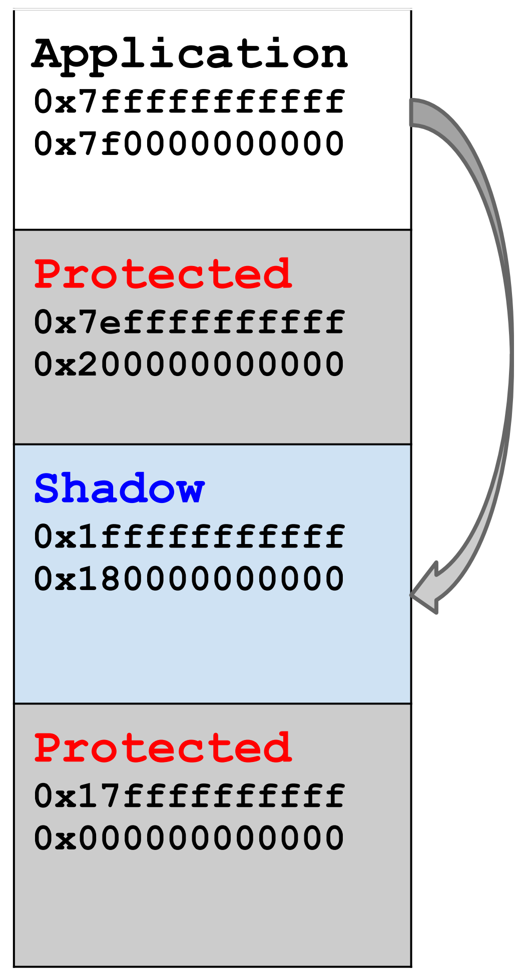
Compiler instrumentation

```
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 & Mask);` // Requires -pie

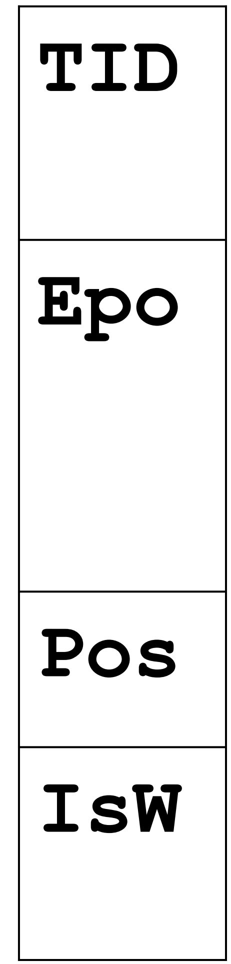


Shadow cell

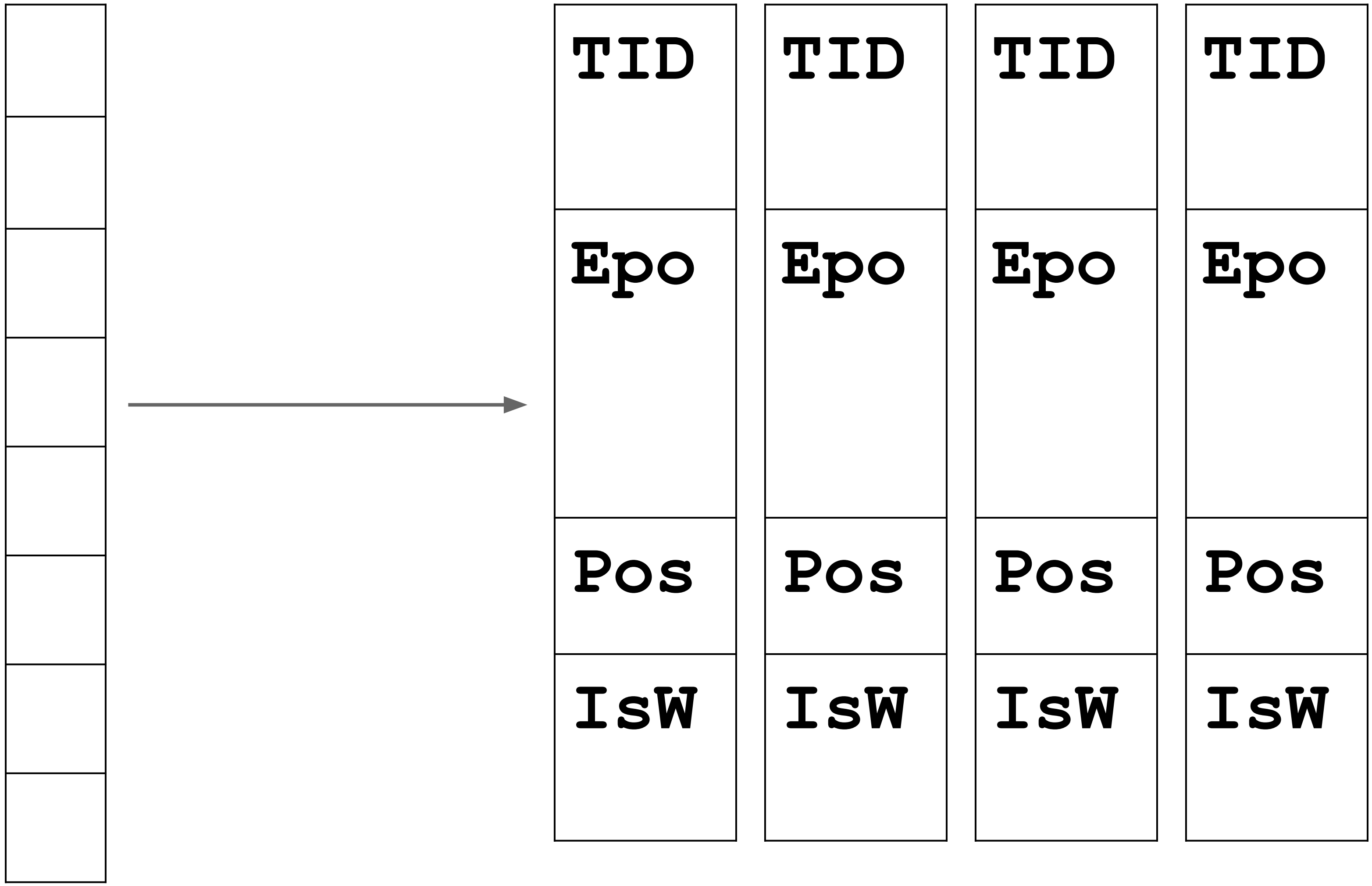
An 8-byte shadow cell represents one memory access:

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

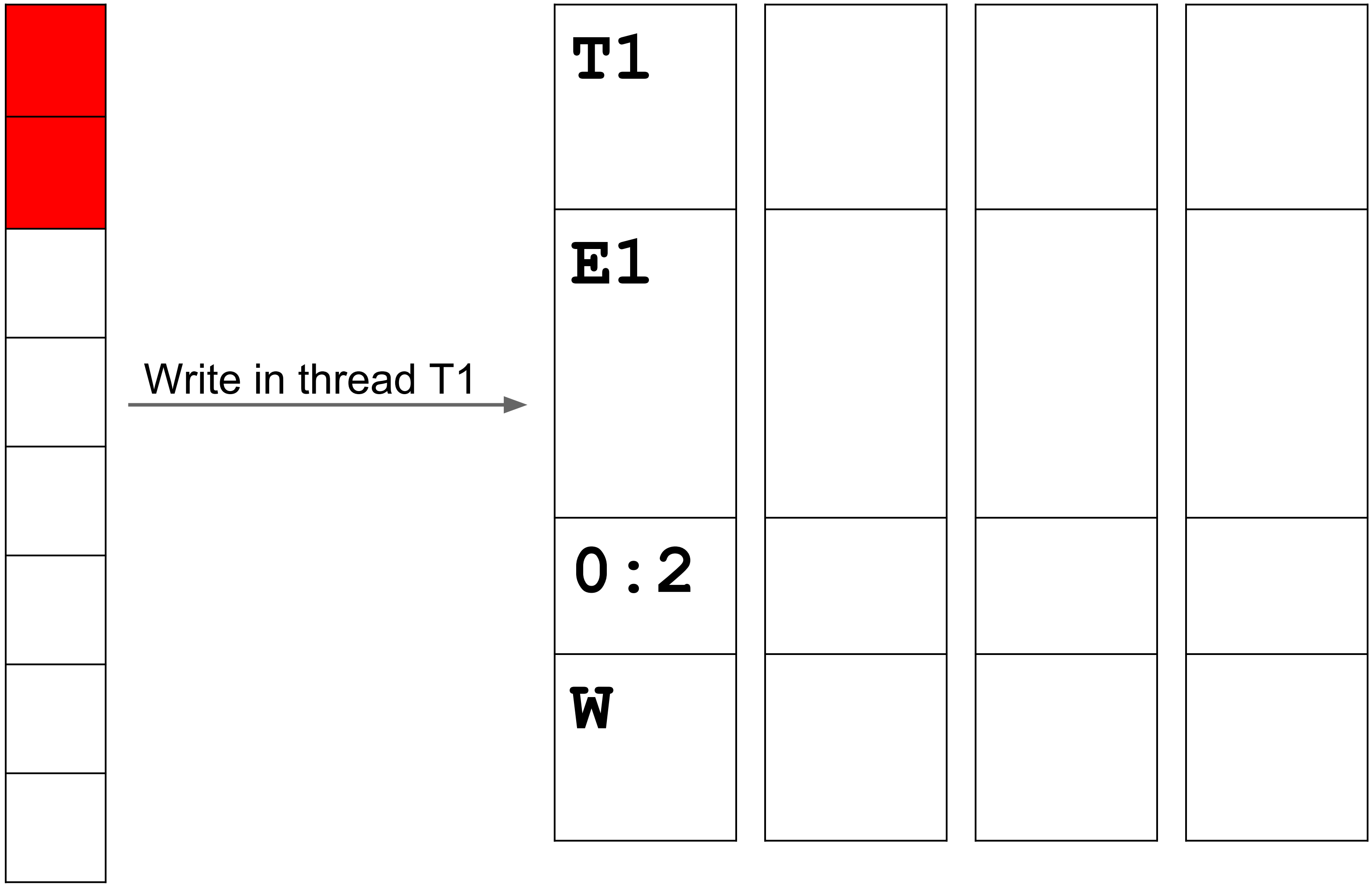
Completely embedded (no more dereferences)



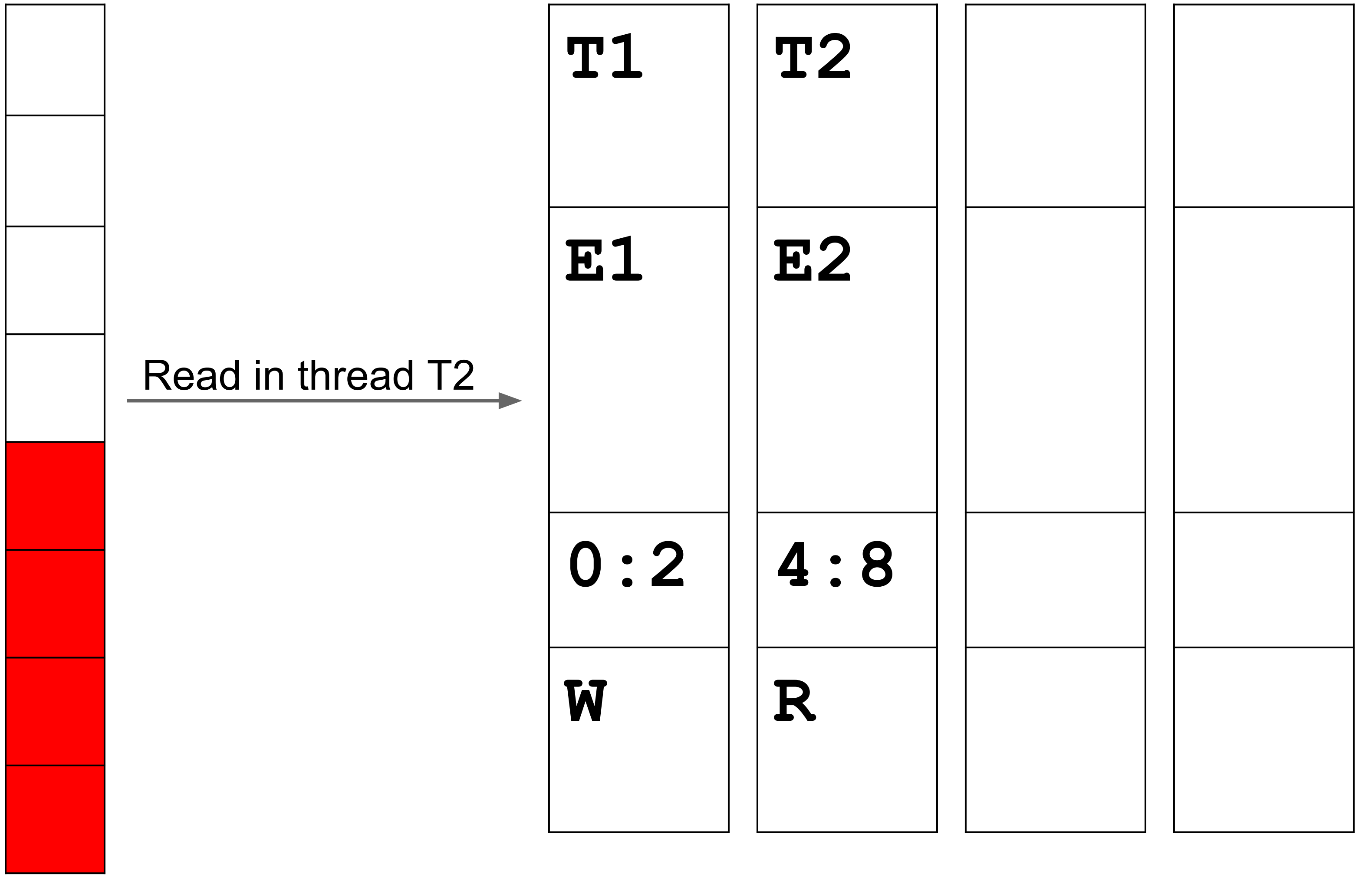
N shadow cells per 8 application bytes



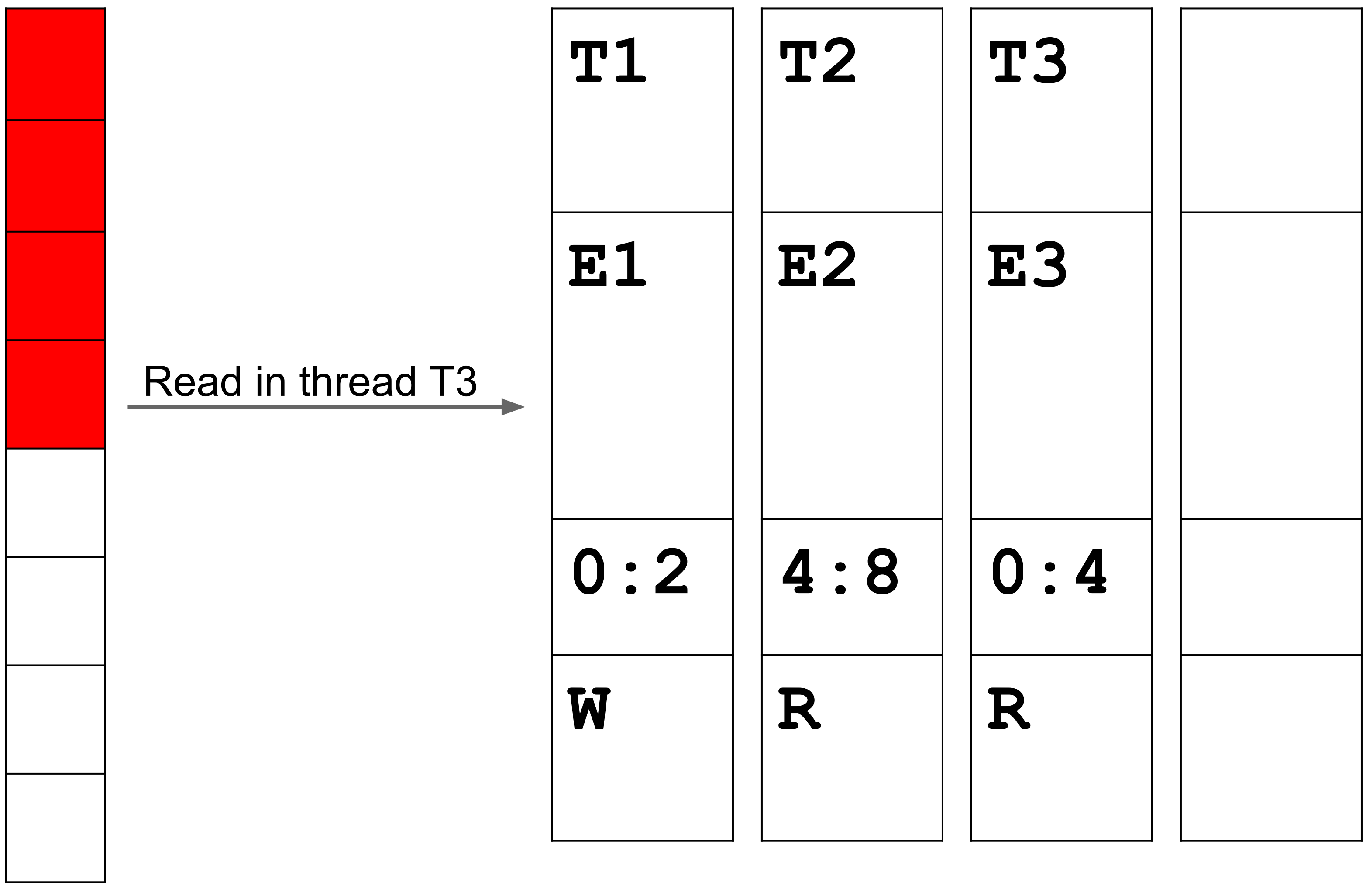
Example: first access



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	

Fast happens-before

- Constant-time operation
  - Get TID and Epoch from the shadow cell
  - 1 load from TLS
  - 1 compare
- Similar to FastTrack (PLDI'09)

Shadow word eviction

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

Informative reports

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

Previous Stack Traces

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

Function interceptors

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

Headaches

- Timeouts
- Memory consumption (OOMs)
- Non-instrumented libraries
- "Benign" data races

AddressSanitizer for Linux Kernel

Disclaimer:
we are not kernel hackers

CONFIG_DEBUG_SLAB

Enables red-zoning and poisoning.

Can detect some out-of-bounds (OOB) accesses and use-after-free (UAF).

Does not detect OOB reads.

Best-effort UAF detection.

CONFIG_KMEMCHECK

CONFIG_KMEMCHECK is a heavy-handed uninitialized memory access checker which causes page-fault on every memory access.

Slow.

CONFIG_DEBUG_PAGEALLOC

Unmaps freed pages from address space. Can detect some UAF accesses.

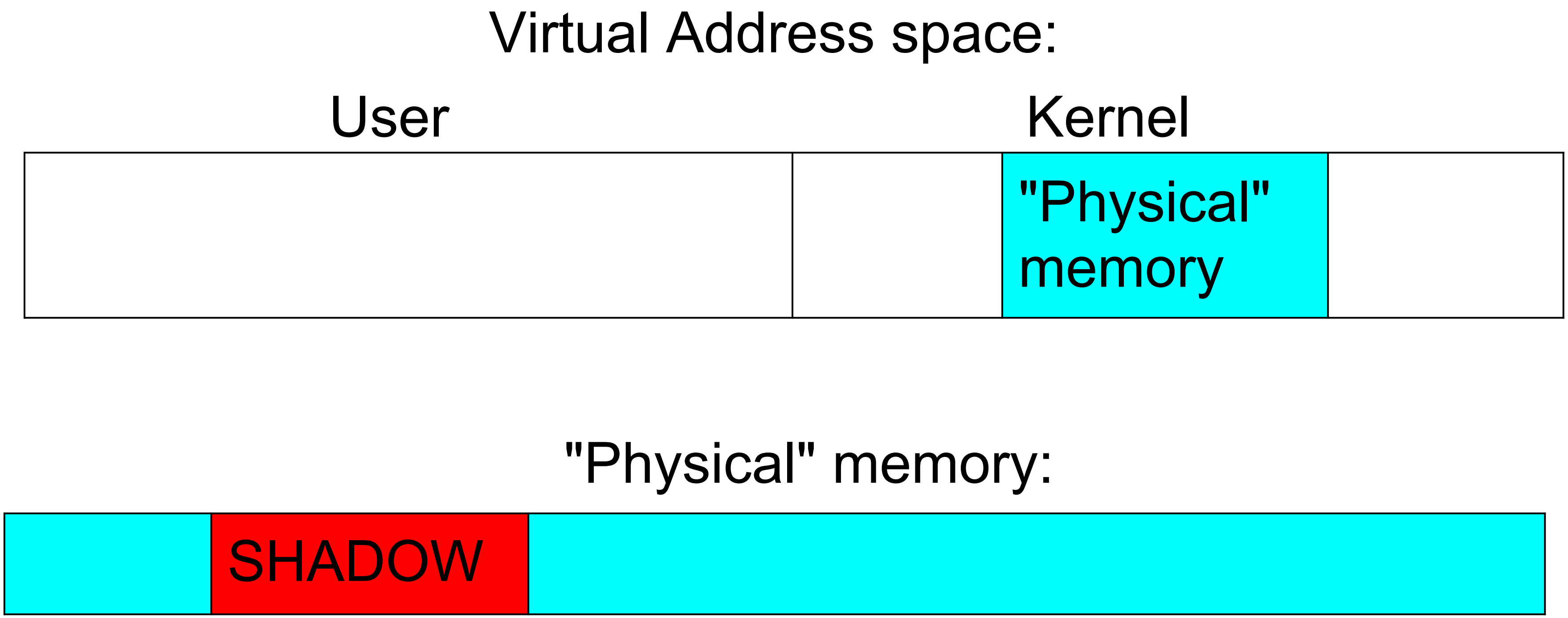
Detects UAF only when the whole page is unused.

CONFIG_ASAN

Fast and comprehensive solution for UAF and OOB.

- Based on compiler instrumentation (fast)
- OOB for both writes and reads
- Strong UAF detection
- Prompt detection of bad memory accesses
- Informative reports

Shadow Memory



Starts at fixed offset (say, 64M)
Size = 1/8 of physical memory

Shadow Mapping

```
char *get_shadow(void *addr) {
  // Is physical memory?
  if (addr < __va(0) ||
      addr > __va(max_pfn << PAGE_SHIFT))
    return NULL;

  return addr / 8 + ASAN_SHADOW_START;
}
```

Virtual Memory?

Unclear what is the best way to handle.

Want the mapping to be (w/o "if" for phys mem):

```
return addr / 8 + ASAN_SHADOW_START;
```

Slab Allocator

- Add redzones
- Poison/unpoison
- Delay reuse (quarantine)

API

asan_poison(addr, size);
asan_unpoison(addr, size);
asan_check(addr, size);

Instrument:

- memset/memcmp/...
- other allocators
- ...

Problems that we know about

- Fast shadow mapping that supports physical/virtual/user memory
- Bootstrap process (instrumentation can't be turned off)
- Text size increase
- Interrupts
- Modules
- ?

ThreadSanitizer for Kernel

ASan needs to intercept **some** memory management + **some** memory accesses.

TSan needs to intercept **all** synchronization + does not tolerate **"benign" races**.

```
int i = atomic_load(&g_index, acquire);

vs

int i = g_index.
rmb();
```

Our requests to
the Kernel and Linux distributions

Ideal Address Space Layout for ASan/TSan/MSan

Everything resides in upper 1/8-th of AS

`0x700000000000-0x7fffffffffff`

Today this works on Linux when:

- x86_64
- -pie
- ASLR on
- Limited stack

Ideally: always

Bill Gates: "16Tb ought to be enough for anybody" (~1981)

Address space with ASLR (0x55555...)

```
int main() { printf("&main: %p\n", &main); }
% setarch x86_64 -R ./a.out

# On Ubuntu 12.04
&main: 0x55555555472c # Breaks TSan mapping

# On Ubuntu 10.04
&main: 0x7ffff7ffe77c # 0x700000000000+ is ok
```

Guilty commit : [2011-11-02]() by Jiri Kosina

Based on original patch by H.J. Lu and Josh Boyer

```
-                    load_bias = 0;
+                    load_bias = ELF_PAGESTART(ELF_ET_DYN_BASE - vaddr);
```

Unlimited stack is too greedy

```
# ulimit -s 8192 # default
00400000-00401000             /tmp/a.out ...
7fed6011a000-7fed6011c000     ld-2.15.so
7fff13a6b000-7fff13a8c000     [stack] ...

# ulimit -s unlimited
00400000-00401000             /tmp/a.out ...
2b86b32e6000-2b86b32e8000     ld-2.15.so
7fff13a6b000-7fff13a8c000     [stack] ...
```

Can you really have 84Tb of stack??

Volatile ranges for shadow memory

In ASan/MSan/TSan's: shadow value '0' means 'good'.

If the process is short of RAM, LRU shadow pages may be confiscated.

Empty pages will be returned on next access.

Was independently proposed as fadvise(FADV_VOLATILE)

How to limit the real memory?

- ulimit -v is useless
  - ASan uses 20Tb of AS
  - TSan uses 97Tb of AS
  - MSan uses 72Tb of AS

General robustness with MAP_NORESERVE

- Conflicts with mlockall(MCL_CURRENT)
  - kills the machine
  - fixed on Feb 2013

- OOMs often kill the machine

Shipping instrumented libraries with Linux distros

- ASan/TSan/MSan use compiler instrumentation
  - Finding buggy accesses in popular libc functions using interceptors
  - Not finding buggy accesses in other standard libs
  - TSan: may cause false positives if libs have synchronization using raw atomics

- Solution: ship instrumented libraries with Linux distros

Summary

- AddressSanitizer
  - Finds buffer overflows and use-after-free
  - "Must have" for all C/C++ developers

- ThreadSanitizer
  - Finds data races
  - "Must have" for all C/C++/Go developers w/ threads

- AddressSanitizer is possible for the Kernel
  - In the investigation stage, help is welcome

- Support from Kernel and Linux distros may help Sanitizers get better

Q&A

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

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

Backup

AddressSanitizer vs Valgrind (Memcheck)

	Valgrind	AddressSanitizer
Heap out-of-bounds	YES	YES
Stack out-of-bounds	NO	YES
Global out-of-bounds	NO	YES
Use-after-free	YES	YES
Use-after-return	NO	Sometimes/YES
Uninitialized reads	YES	NO
Overhead	10x-300x	1.5x-3x
Platforms	Linux, Mac	Same as LLVM *