AddressSanitizer: A Fast Address Sanity Checker

Konstantin Serebryany, Derek Bruening,
Alexander Potapenko, Dmitry Vyukov

USENIX ATC '2012
June 15, 2012

Google™

Memory Bugs in C++

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

Google™

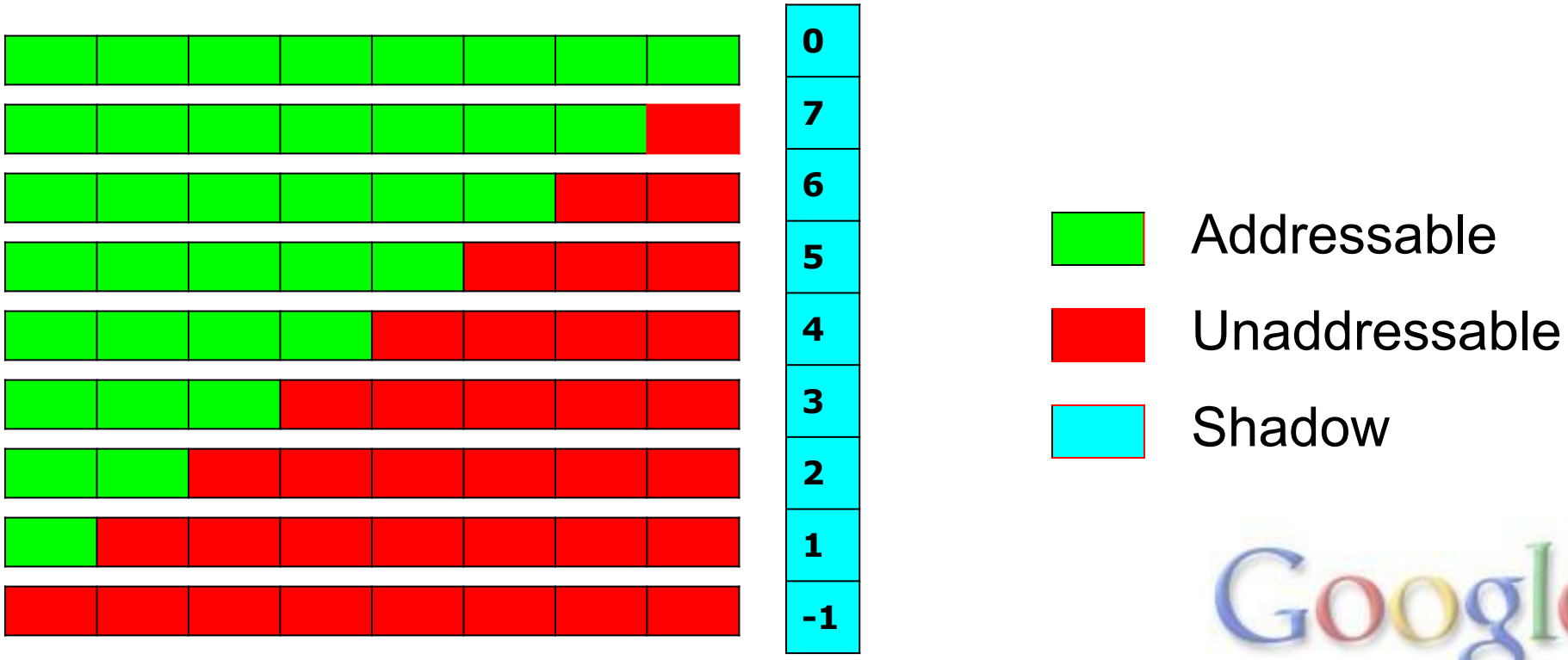
AddressSanitizer overview

- Compile-time instrumentation module
  - LLVM, platform independent
  - ~1 KLOC
- Run-time library
  - Supports Linux, MacOS, Android
  - ~9 KLOC
- Released in May 2011
- Part of LLVM since November 2011

Google™

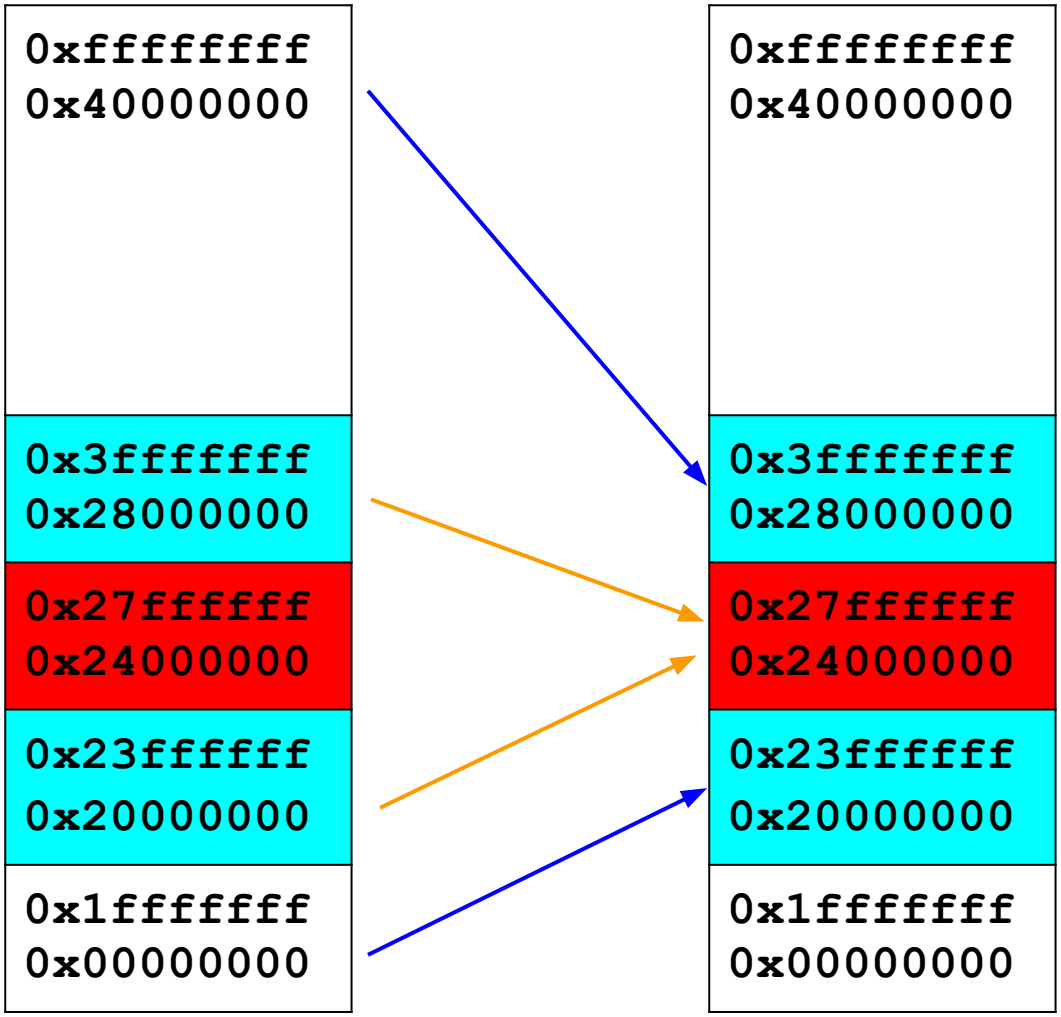
Shadow byte

- 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.



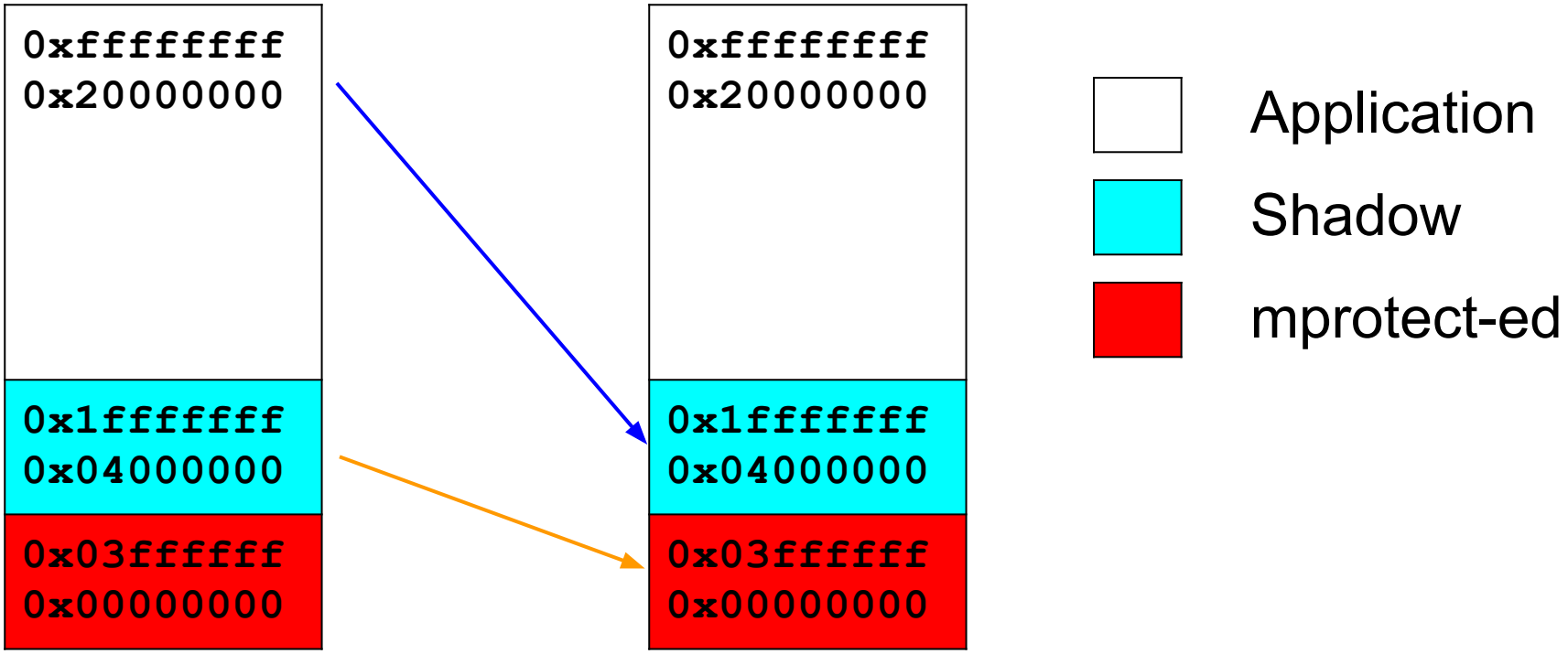
Google

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



Google

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

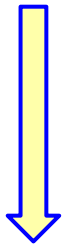


- Requires `-fPIE -pie` (linux)
- Gives ~6% speedup on x86_64

Google

Instrumentation: 8 byte access

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



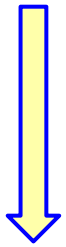
```
char *shadow = (a>>3)+Offset;
if (*shadow)

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

Google

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

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



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

Google

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>
ud2a                    # generate SIGILL*

movq $0x1234,(%rdi) # original store
```

*** May use call instead of UD2**

Instrumenting stack

```
void foo() {

  char a[328];









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

}
```

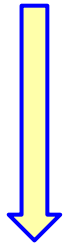
Instrumenting stack

```
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

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

Report example: stack-buffer-overflow

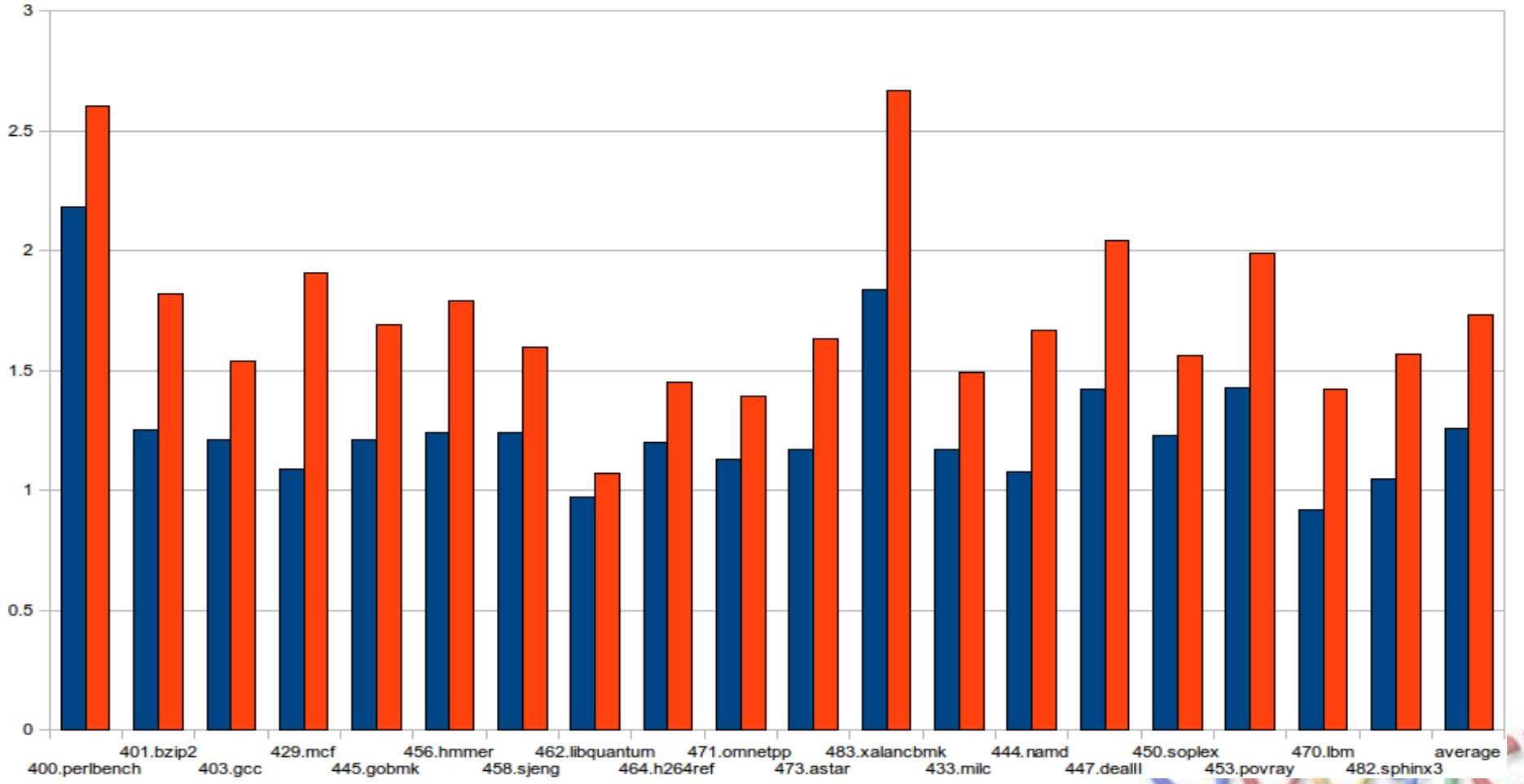
ERROR: AddressSanitizer **stack-buffer-overflow**
  on address 0x7f5620d981b4
  at pc 0x4024e8 bp 0x7fff101cbc90 sp 0x7fff101cbc88

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'

Google™

1.73x slowdown (reads & writes)
1.26x slowdown (writes only)



Real-life performance

- Almost no slowdown for GUI programs
  - They don't consume all of CPU anyway
- 1.5x - 4x slowdown for server side apps with -O2
  - The larger the slower (instruction cache)
  - Up to 8x with -O1 (inlining, instruction cache)

Memory overhead

- Heap redzones
  - default is 128-255 bytes per allocation
  - minimal is 32-63 bytes per allocation
- Stack redzones: 32-63 bytes per address-taken local variable
- Global redzones: 32-63 bytes per global
- Fixed size Quarantine (256M)
- (Heap + Globals + Stack + Quarantine) / 8 for shadow

- **Typical overall memory overhead is 2x - 4x**
  - Seen between 1.1x and 20x
- Stack size increase
  - Seen up to 3x
- Maps (but does not reserve) 1/8-th of all address space
  - 16T 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
- ...

Google™

Future work

- Avoid redundant checks (static analysis)
- Instrument or recompile libraries
- Instrument inline assembler
- Port to Windows
  - Mostly, Clang
  - Plain C and simple C++ already works
  - Help is welcome!
- Implement GCC variant
  - Help is welcome!

Summary

C++ is suddenly
a much safer language

Google™

Implement AddressSanitizer in Hardware

Google™

Q&A

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

Google™

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-30x	1.5x-3x
Platforms	Linux, Mac	Same as LLVM *