KernelAddressSanitizer (KASan)

a fast memory error detector for the Linux kernel

Andrey Konovalov <andreyknvl@google.com>, Google
Dmitry Vyukov <dvyukov@google.com>, Google

LinuxCon North America 2015
July 18th 2015

Agenda

- Userspace tools
- KernelAddressSanitizer (KASan)
- KernelThreadSanitizer (KTSan)
- Requests and suggestions
- Future plans

Userspace tools

- AddressSanitizer (ASan)
  - detects use-after-free and out-of-bounds
- ThreadSanitizer (TSan)
  - detects data races and deadlocks
- MemorySanitizer (MSan)
  - detects uninitialized memory uses
- UndefinedBehaviorSanitizer (UBSan)
  - detects undefined behaviors in C/C++

KernelAddressSanitizer (KASan)

Userspace ASan

AddressSanitizer - a fast memory error detector for C/C++

- Finds
  - Buffer overflows in heap, stack and globals
  - heap-use-after-free, stack-use-after-return
- Status
  - Linux / OSX / Windows / Android / FreeBSD / iOS / ...
  - The average slowdown is ~2x
  - The average memory overhead is ~2-3x
  - 10000+ bugs found (Chromium, Firefox, …)
- Easy to use
  - $ gcc -fsanitize=address main.c
  - $ clang -fsanitize=address main.c

ASan report example

```
$ cat a.cc
int main(int argc, char **argv) {
   int *array = new int[100];
   delete[] array;
   return array[argc];  // BOOM
}

$ clang++ -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
```

Kernel memory debugging

- SLUB_DEBUG / DEBUG_SLAB
  - Enables redzones and poisoning (writing magic values to check later)
  - Can detect some out-of-bounds and use-after-free accesses
  - Can’t detect out-of-bounds reads
  - Detects bugs only on allocation / freeing in some cases
- DEBUG_PAGEALLOC
  - Unmaps freed pages from address space
  - Can detect some use-after-free accesses
  - Detects use-after-free only when the whole page is unused
- kmemcheck
  - Detects use-after-free accesses and uninitialized-memory-reads
  - Causes page fault on each memory access (slow)

KASan

- Fast and comprehensive solution for both UAF and OOB
  - Based on compiler instrumentation
  - Detects out-of-bounds for both writes and reads
  - Has strong use-after-free detection
  - Detects bugs at the point of occurrence
  - Prints informative reports

Two parts

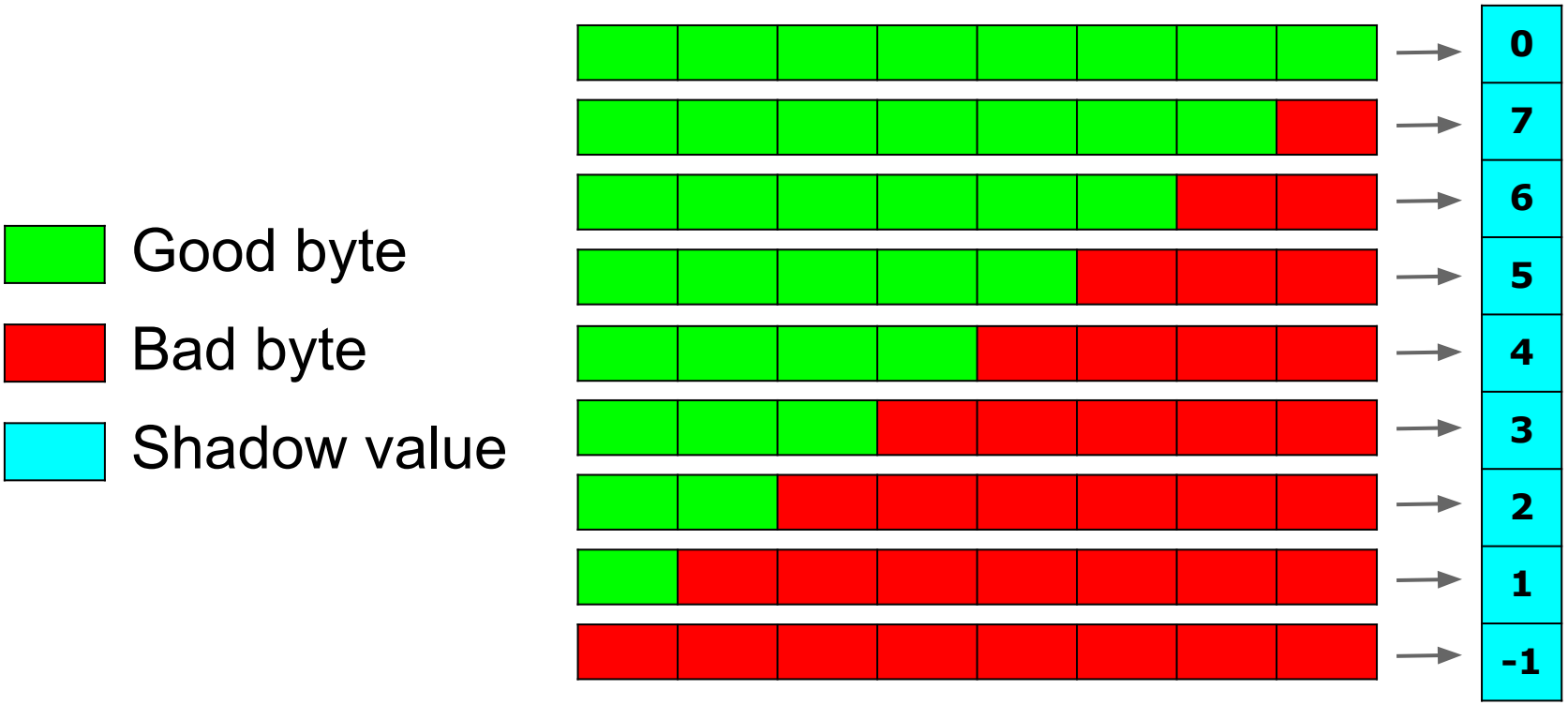
- Compiler module
  - Instruments memory accesses

- Runtime part
  - Bug detection algorithm

Shadow byte

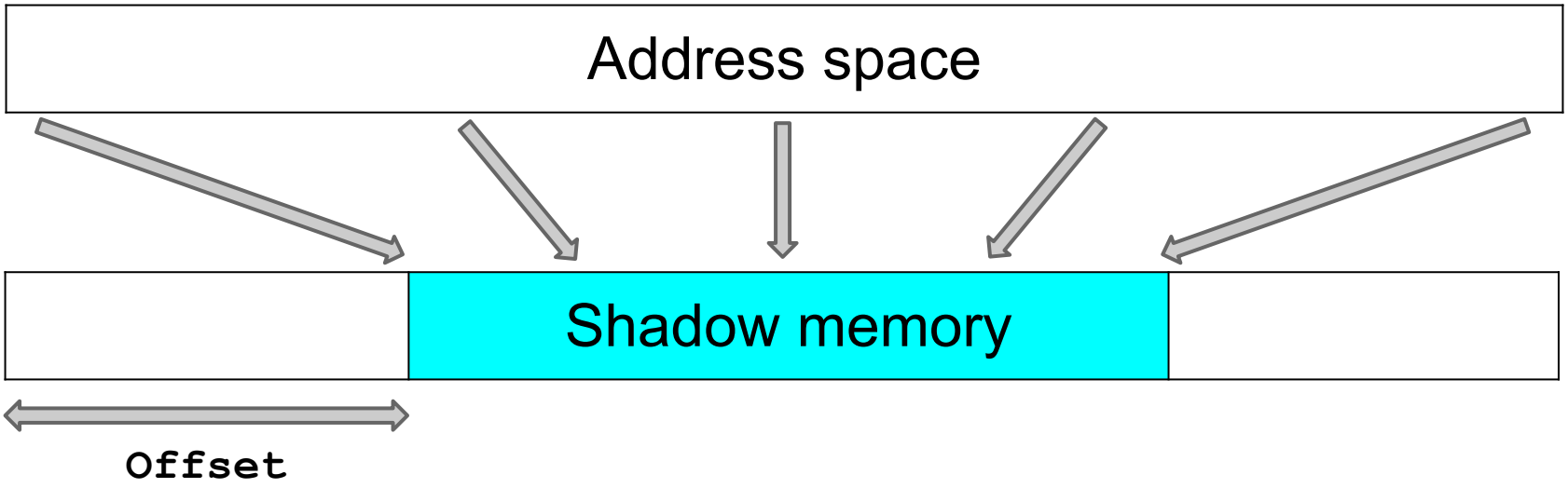
Any aligned 8 bytes may have 9 states:

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



Memory mapping

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

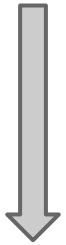


x86-64 memory layout

```
0000000000000000 - 00007fffffffffff (=47 bits) user space, different per mm
hole caused by [48:63] sign extension
ffff800000000000 - ffff87ffffffffff (=43 bits) guard hole, reserved for hypervisor
ffff880000000000 - ffffc7ffffffffff (=64 TB) direct mapping of all phys. memory
ffffc80000000000 - ffffc8ffffffffff (=40 bits) ho le
ffffc90000000000 - ffffe8ffffffffff (=45 bits) vmalloc/ioremap space
ffffe90000000000 - ffffe9ffffffffff (=40 bits) hole
ffffea0000000000 - ffffeaffffffffff (=40 bits) virtual memory map (1TB)
... unused hole ...
ffffec0000000000 - fffffc0000000000 (=44 bits) kasan shadow memory (16TB)
... unused hole ...
ffffff0000000000 - ffffff7fffffffff (=39 bits) %esp fixup stacks
... unused hole ...
ffffffff80000000 - ffffffffa0000000 (=512 MB)  kernel text mapping, from phys 0
ffffffffa0000000 - fffffffff5ffffff (=1525 MB) module mapping space
ffffffffff600000 - ffffffffffdfffff (=8 MB) vsyscalls
ffffffffffe00000 - ffffffffffffffff (=2 MB) unused hole
```

Compiler instrumentation: 8 byte access

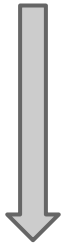
```
*a = ...
```



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

Stack instrumentation

```
void foo() {

 char a[328];









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

}
```

Stack instrumentation

```
void foo() {
 char rz1[32]; // 32-byte aligned
 char a[328];









}
```

Stack instrumentation

```
void foo() {
 char rz1[32]; // 32-byte aligned
 char a[328];
 char rz2[24];
 char rz3[32];









}
```

Stack instrumentation

```
void foo() {
 char rz1[32]; // 32-byte aligned
 char a[328];
 char rz2[24];
 char rz3[32];
 int *shadow = (&rz1 >> 3) + kOffset;









}
```

Stack instrumentation

```
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




}
```

Stack instrumentation

```
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



}
```

Stack instrumentation

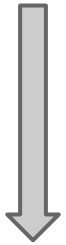
```
void foo() {
 char rz1[32]; // 32-byte aligned
 char a[328];
 char rz2[24];
 char rz3[32];
 int *shadow = (&rz1 >> 3) + kOffset;
 shadow[0] = 0xffffffff; // poison rz1
 shadow[11] = 0xffffff00; // poison rz2
 shadow[12] = 0xffffffff; // poison rz3
 <------------ CODE ------------>

}
```

Stack instrumentation

```
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;
}
```

Globals instrumentation

```
int a;
```



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

Runtime part

- Maps shadow memory
- Enables KASan
- Allocator extensions
  - Poison/unpoison memory on each kfree/kmalloc
  - Add poisoned redzones around slab objects
  - Put freed slab objects in a delayed reuse queue
  - Collect stack traces on each kmalloc/kfree
- Prints error messages

KASan shadow mapping

- Shadow mapped to a zero page on early stage

- Map real shadow when page tables are initialized
  - Physical memory
  - Kernel text mapping

- Map shadow for modules when they are loaded

Slab layout

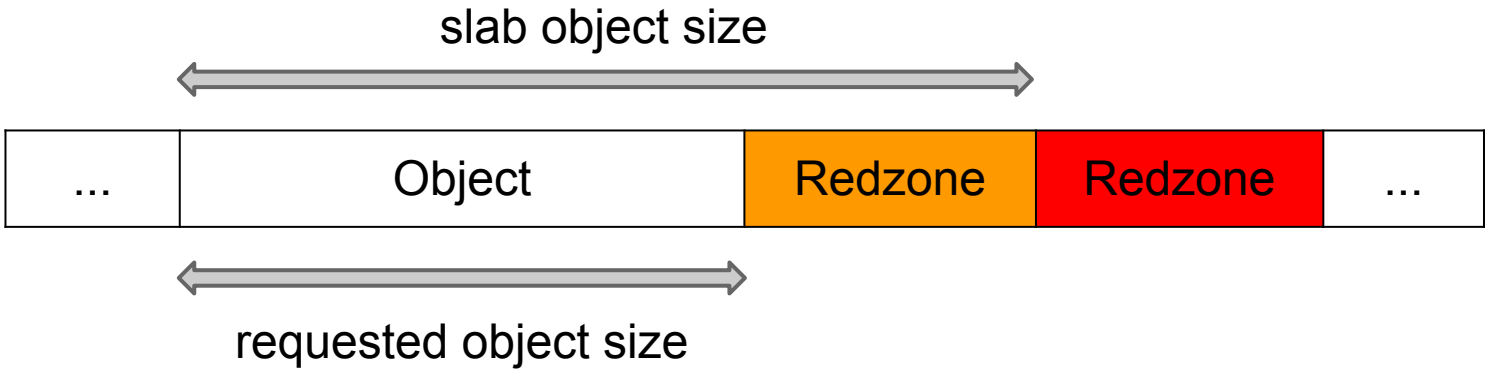
- The whole slab is poisoned when created
- Objects are unpoisoned when allocated
- 32-bit handles to the allocation and deallocation stacks are stored in the redzones

Usual slab layout:

Slab layout with KASan:

kmalloc slabs

- kmalloc slabs have power-of-two sized objects
- If kmalloc requests **`N`** bytes and slab with object size of **`K`** is used, the last (**`K - N`**) bytes are poisoned



Quarantine

- Freed objects are put into a delayed reuse queue

- Higher chance to trigger use-after-free

Not instrumented

- Early code
  - Before early shadow is mapped
- Allocator
  - May validly access slab metadata
- Assembly
  - Not supported by the compiler

- Binary modules have to be rebuilt with KASan

User-memory-access bug

- When the kernel accesses user space memory without using special API (copy_to_user / copy_from_user)

- Specific to the kernel

- Detected by KASan
  - shadow is not mapped for the user address space
  - page fault happens

Example

```
int example(void)
{
    size_t size = sizeof(int) * 100;
    int *array = (int *)kmalloc(size, GFP_KERNEL);
    // kasan_unpoison_shadow(array, size);

    // Validly using array here.

    kfree(array);
    // kasan_poison_shadow(array, size);

    // void *addr = &array[42];
    // char *shadow = (addr >> 3) + Offset;
    // if (*shadow && *shadow < (addr & 7) + 4)
    //     ReportError(addr);
    return array[42];
}
```

Report example

```
AddressSanitizer: heap-buffer-overflow on address ffff8800205f0e40
Write of size 1 by thread T14005:
 [<ffffffff811e2542>] ftrace_event_write+0xe2/0x130 ./kernel/trace/trace_events.c:583
 [<ffffffff8128c497>] vfs_write+0x127/0x2f0 ??:0
 [<ffffffff8128d572>] SyS_write+0x72/0xd0 ??:0
 [<ffffffff818423d2>] system_call_fastpath+0x16/0x1b ./arch/x86/kernel/entry_64.S:629

Allocated by thread T14005:
 [<     inlined    >] trace_parser_get_init+0x28/0x70 kmalloc ./include/linux/slab.h:413
 [<ffffffff811cc258>] trace_parser_get_init+0x28/0x70 ./kernel/trace/trace.c:759
 [<ffffffff811e24d2>] ftrace_event_write+0x72/0x130 ./kernel/trace/trace_events.c:572
 [<ffffffff8128c497>] vfs_write+0x127/0x2f0 ??:0
 [<ffffffff8128d572>] SyS_write+0x72/0xd0 ??:0
 [<ffffffff818423d2>] system_call_fastpath+0x16/0x1b ./arch/x86/kernel/entry_64.S:629

The buggy address ffff8800205f0e40 is located 0 bytes to the right
 of 128-byte region [ffff8800205f0dc0, ffff8800205f0e40)
```

Report example, continued

```
Memory state around the buggy address:
 ffff8800205f0900: rrrrrrrr rrrrrrrr rrrrrrrr rrrrrrrr
 ffff8800205f0a00: rrrrrrrr ........ ........ rrrrrrrr
 ffff8800205f0b00: rrrrrrrr rrrrrrrr rrrrrrrr rrrrrrrr
 ffff8800205f0c00: ........ .......5 rrrrrrrr rrrrrrrr
 ffff8800205f0d00: rrrrrrrr rrrrrrrr rrrrrrrr ........
>ffff8800205f0e00: ........ rrrrrrrr rrrrrrrr rrrrrrrr
                            ^
 ffff8800205f0f00: rrrrrrrr rrrrrrrr rrrrrrrr rrrrrrrr
 ffff8800205f1000: ........ ........ ........ ........
 ffff8800205f1100: ........ ........ ........ ........
 ffff8800205f1200: ........ ........ ........ ........
 ffff8800205f1300: ........ ........ ........ ........
Legend:
 f - 8 freed bytes
 r - 8 redzone bytes
 . - 8 allocated bytes
 x=1..7 - x allocated bytes + (8-x) redzone bytes
```

Trophies

- 65 bugs found so far
  - 35 use-after-free
  - 18 heap-out-of-bounds
  - 8 stack-out-of-bounds
  - 2 global-out-of-bounds
  - 2 user-memory-access

- CVE-2013-4387
  - Remote Denial-Of-Service

KASan vs kmemcheck

	kmemcheck	KASan
buffer-overflow in heap, stack and globals	-	+
use-after-free	+-	+
uninitialized-memory-read	+	-
user-memory-access	-	+
slowdown	~10x	~1.5x
memory usage	~2x	~2x

KASan status

- CONFIG_KASAN is available upstream since 4.0
  - Thanks to Andrey Ryabinin
  - Supports x86-64, SLUB allocator
  - arm64 on the way

- Some features are not upstream yet
  - Stack depot
  - Quarantine

Using KASan

- KASan is a dynamic detector, which means a bug is detected when it actually occurs

- Tests with good coverage
- Fuzzing (Trinity, iknowthis, perf_fuzzer)

KernelThreadSanitizer (KTSan)

Data race

- A **data race** occurs when two threads access the same variable concurrently and at least one of the accesses is a write.

Userspace TSan

ThreadSanitizer - a fast data race detector for C/C++ and Go

- Status
  - C++: Linux / FreeBSD
  - Go: Linux / Mac / Windows / FreeBSD
  - The average slowdown is ~5x
  - The average memory overhead is ~5-10x
  - 1000+ bugs found

- Easy to use
  - `$ gcc -fsanitize=thread main.c`
  - `$ clang -fsanitize=thread main.c`
  - `$ go run -race main.go`

KTSan status

- Prototype available
  - Work in progress
  - x86-64 only

- Found it’s first harmful data race a few weeks ago!

KTSan report example

```
ThreadSanitizer: data-race in SyS_swapon

Read of size 8 by thread T307 (K7621):
 [<     inlined    >] SyS_swapon+0x3c0/0x1850 SYSC_swapon mm/swapfile.c:2395
 [<ffffffff812242c0>] SyS_swapon+0x3c0/0x1850 mm/swapfile.c:2345
 [<ffffffff81e97c8a>] ia32_do_call+0x1b/0x25 arch/x86/entry/entry_64_compat.S:500

Previous write of size 8 by thread T322 (K7625):
 [<     inlined    >] SyS_swapon+0x809/0x1850 SYSC_swapon mm/swapfile.c:2540
 [<ffffffff81224709>] SyS_swapon+0x809/0x1850 mm/swapfile.c:2345
 [<ffffffff81e957ae>] entry_SYSCALL_64_fastpath+0x12/0x71 arch/x86/entry/entry_64.S:186
```

- Forgotten mutex
- Can lead to a very hard to debug racy use-after-free bugs

Issues

- Benign data races

- Inconsistent kernel atomics API

Benign data races in the kernel

- GCC used to guarantee that machine-word-sized accesses would be atomic
  - not anymore
  - a lot of kernel code relies on this
  - a lot of benign races as a result

- Benign data races
  - cause undefined behavior according to the new standard
  - do not allow any formal verification

- Lots of reports
- Easy to miss real races

Kernel atomics API

- atomic_set / atomic_load / atomic_add / atomic_inc / ...
  - relaxed
- xchg / cmpxchg / atomic_xchg / atomic_cmpxchg / atomic_inc_return / …
  - release-acquire
- xadd
  - not documented at all
  - release-acquire based on the source
- WRITE_ONCE / READ_ONCE (ACCESS_ONCE)
  - documented as macros to prevent compiler reordering
  - used as relaxed stores / loads
- smp_store_release / smp_load_acquire
  - used as store-release / load-acquire

Requests & Suggestions

- Fix benign races
  - to avoid C undefined behavior
  - to make synchronization more visible
  - to allow formal verification

- Consistent atomic API
  - a very big task
  - C11 atomic API
    - atomic_load(addr, memory_order_relaxed)
    - atomic_store(addr, value, memory_order_release)
  - to improve code readability

Future plans

- KASan
  - Upstream more features
  - Other archs (x86-32, arm32, arm64)
  - Use-after-return, use-after-scope

- KTSan

- KMSan (KernelMemorySanitizer)
  - KASan + KMSan = feature parity with kmemcheck

- Coverage guided fuzzer

Summary

- KernelAddressSanitizer
  - Available upstream
  - Found 50+ bugs

- KernelThreadSanitizer
  - Prototype available
  - Already found its first few bugs

- Requests and suggestions
  - Fix benign data races
  - Consistent atomic API

Questions?

https://github.com/google/kasan/wiki
https://github.com/google/ktsan/wiki

Andrey Konovalov, andreyknvl@google.com
Dmitry Vyukov, dvyukov@google.com
Andrey Ryabinin, ryabinin.a.a@gmail.com