Runtime analysis with Valgrind and Google Sanitizers

Tom Peters, August 22, 2018
thomas.d.peters@gmail.com
`tdp2110` on slack, github

Outline

Disclaimer: this talk will be Linux-heavy.

1. Valgrind and valgrind tools (mostly memcheck)
2. Sanitizers (asan, msan, tsan, ubsan)
3. Practical recommendations

Remember: C++ is dangerous

- Null pointer dereference
- Signed integer overflow
- Shifting beyond width
- Array access out of bounds
- Modification of a const object
- Integer division by zero
- Data races
- Reaching end of value-returning function (other than main) without returning
- Strict aliasing violations
- ODR violation
- … (the list goes on and on)

Tools to fight Undefined Behavior

Runtime (aka dynamic) analysis

- Refers to catching bugs as they happen in an application

Static analysis

- Looks for bugs "without executing the program"

Why dynamic analysis

Catch real bugs as they happen in the wild.

Weaknesses of static analysis

- Big difference between free and proprietary tools
- Proprietary tools are expensive
- Sometimes hard to use (or at least need hand-holding)
- Human problem: often easy to ignore.

Weaknesses of dynamic analysis

- You must hit the bug. (a symptom of being "unsound")
- Sometimes "too late": compiler may have,eg, deleted code.

Valgrind (http://valgrind.org/)

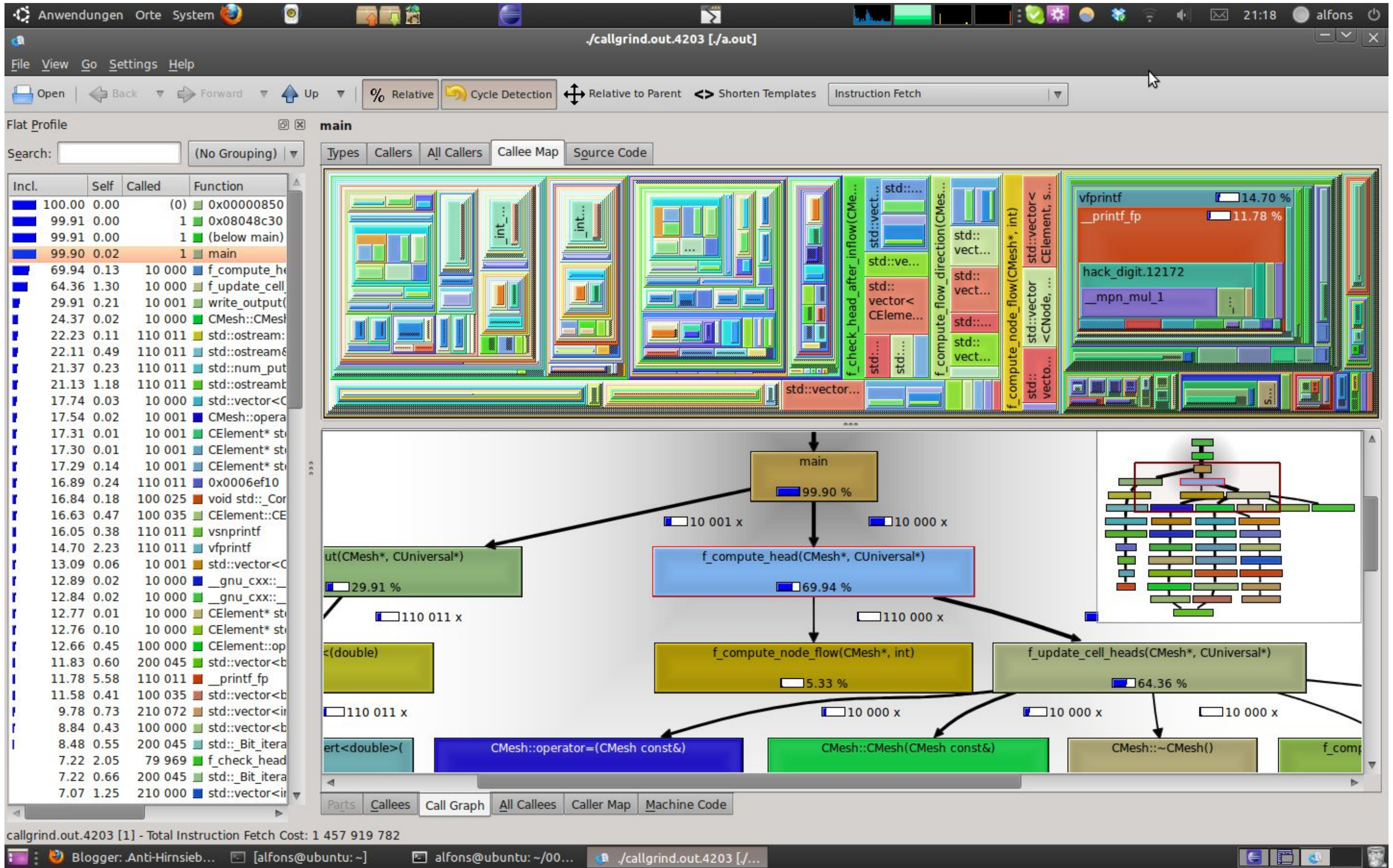
A framework for dynamic binary instrumentation.

"Tools" use the framework to do something interesting with a binary.

Existing Valgrind tools:

- **Memcheck**: a memory checker.
- **Cachegrind**: a CPU cache profiler. It simulates instruction cache, and data caches.
- **Callgrind**: extension of cachegrind to include call tree information
- **Massif**: heap profiler
- **Helgrind**: a thread debugger

kcachegrind



Incl.	Self	Called	Function
100.00	0.00	(0)	0x00000850
99.91	0.00	1	0x08048c30
99.91	0.00	1	(below main)
99.90	0.02	1	main
69.94	0.13	10 000	f_compute_he
64.36	1.30	10 000	f_update_cell_
29.91	0.21	10 001	write_output(
24.37	0.02	10 000	CMesh::CMesh
22.23	0.11	110 011	std::ostream:
22.11	0.49	110 011	std::ostream&
21.37	0.23	110 011	std::num_put
21.13	1.18	110 011	std::ostreamb
17.74	0.03	10 000	std::vector<C
17.54	0.02	10 001	CMesh::opera
17.31	0.01	10 001	CElement* st
17.30	0.01	10 001	CElement* st
17.29	0.14	10 001	CElement* st
16.89	0.24	110 011	0x0006ef10
16.84	0.18	100 025	void std::_Cor
16.63	0.47	100 035	CElement::CE
16.05	0.38	110 011	vsnprintf
14.70	2.23	110 011	vfprintf
13.09	0.06	10 001	std::vector<C
12.89	0.02	10 000	__gnu_cxx::_
12.84	0.02	10 000	__gnu_cxx::_
12.77	0.01	10 000	CElement* st
12.76	0.10	10 000	CElement* st
12.66	0.45	100 000	CElement::op
11.83	0.60	200 045	std::vector<b
11.78	5.58	110 011	__printf_fp
11.58	0.41	100 035	std::vector<b
9.78	0.73	210 072	std::vector<ir
8.84	0.43	100 000	std::vector<b
8.48	0.55	200 045	std::_Bit_itera
7.22	2.05	79 969	f_check_head
7.22	0.66	200 045	std::_Bit_itera
7.07	1.25	210 000	std::vector<ir

callgrind.out.4203 [1] - Total Instruction Fetch Cost: 1 457 919 782

Memcheck

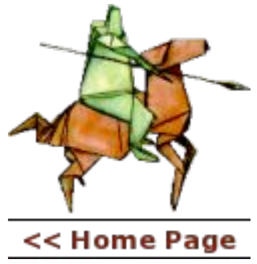
Memory checker built off valgrind-core

- **Addressability** - heap-buffer overflow, use after free, access beyond the top of the stack
- **(bitwise) Validity** - use of uninitialized memory (sometimes with bit-precision)
- Heap **bookkeeping**: Memory leaks, mismatched new[]/delete, double free
- Other goodies: Detection of overlapping arguments to memcpy

memcheck

"Keeping with the Nordic theme, Valgrind was chosen. Valgrind is the name of the main entrance to Valhalla (the Hall of the Chosen Slain in Asgard). Over this entrance there resides a wolf and over it there is the head of a boar and on it perches a huge eagle, whose eyes can see to the far regions of the nine worlds. Only those judged worthy by the guardians are allowed to pass through Valgrind. All others are refused entrance."

Julian Seward (worked on GHC, works at Mozilla)
Nicholas Nethercote (now works at Mozilla)
First release 2002, worked on to this day



Memcheck Demo

Memcheck Supported platforms

- x86/Linux: up to and including SSSE3, but not higher -- no SSE4, AVX, AVX2. This target is in maintenance mode now..
- AMD64/Linux: up to and including AVX2. This is the primary development target and tends to be well supported.
- PPC32/Linux, PPC64/Linux, PPC64LE/Linux: up to and including Power8.
- S390X/Linux: supported.
- ARM/Linux: supported since ARMv7.
- ARM64/Linux: supported for ARMv8.
- MIPS32/Linux, MIPS64/Linux: supported.
- X86/Solaris, AMD64/Solaris, X86/illumos, AMD64/illumos: supported since Solaris 11.
- X86/Darwin (10.10, 10.11), AMD64/Darwin (10.10, 10.11): supported.
- ARM/Android, ARM64/Android, MIPS32/Android, X86/Android: supported.

Valgrind Core

- Executes client ("guest") code indirectly in a sort of VM
    - ThreadState data structure holds client registers (and "shadow" registers)
- Disassembles chunks of client binary into intermediate representation (IR)
    - Disassembled code pulls guest registers from ThreadState
- IR is passed to tools (such as memcheck), which modify it to do something else
- Instrumented IR is optimized, JITted, and then executes it.
- Maintains code cache, finds, creates, and executes translations.
- Thread safe, but has a global interpreter lock

Valgrind core

Valgrind uses "disassemble and resynthesize" (D&R)

Other DBI frameworks use "copy and annotate"

Translation phases

1. Disassembly*: machine code → tree IR.
2. Optimization 1: tree IR → flat IR.
3. Instrumentation: flat IR → flat IR.
4. Optimization 2: flat IR → flat IR.
5. Tree building: flat IR → tree IR.
6. Instruction selection*: tree IR → instruction list.
7. Register allocation: instruction list → instruction list.

example: machine code → tree IR

```
0x24F275: movl -16180(%ebx,%eax,4),%eax
 1: ------ IMark(0x24F275, 7) ------
 2: t0 = Add32(Add32(GET:I32(12), # get %ebx and
      Shl32(GET:I32(0),0x2:I8)),  # %eax, and
      0xFFFFC0CC:I32)             # compute addr
 3: PUT(0) = LDle:I32(t0)         # put %eax

0x24F27C: addl %ebx,%eax
 4: ------ IMark(0x24F27C, 2) ------
 5: PUT(60) = 0x24F27C:I32  # put %eip
 6: t3 = GET:I32(0)         # get %eax
 7: t2 = GET:I32(12)        # get %ebx
 8: t1 = Add32(t3,t2)       # addl
 9: PUT(32) = 0x3:I32       # put eflags val1
10: PUT(36) = t3            # put eflags val2
11: PUT(40) = t2            # put eflags val3
12: PUT(44) = 0x0:I32       # put eflags val4
13: PUT(0) = t1             # put %eax
```

```
0x24F27E: jmp*l %eax
14: ------ IMark(0x24F27E, 2) ------
15: PUT(60) = 0x24F27E:I32         # put %eip
16: t4 = GET:I32(0)                # get %eax
17: goto {Boring} t4
```

memcheck-instrumented, flat IR

```
0x24F275: movl -16180(%ebx,%eax,4),%eax
* 1: ------ IMark(0x24F275, 7) ------
  2: t11 = GET:I32(320)            # get sh(%eax)
* 3: t8 = GET:I32(0)               # *get %eax
  4: t14 = Shl32(t11,0x2:I8)       # shadow shll
* 5: t7 = Shl32(t8,0x2:I8)         # *shll
  6: t18 = GET:I32(332)            # get sh(%ebx)
* 7: t9 = GET:I32(12)              # *get %ebx
  8: t19 = Or32(t18,t14)           # shadow addl 1/3
  9: t20 = Neg32(t19)              # shadow addl 2/3
 10: t21 = Or32(t19,t20)           # shadow addl 3/3
*11: t6 = Add32(t9,t7)             # *addl
 12: t24 = Neg32(t21)              # shadow addl 1/2
 13: t25 = Or32(t21,t24)           # shadow addl 2/2
*14: t5 = Add32(t6,0xFFFFC0CC:I32) # *addl
 15: t27 = CmpNEZ32(t25)           # shadow loadl 1/3
```

```
 16: DIRTY t27 RdFX-gst(16,4) RdFX-gst(60,4) :::
helperc_value_check4_fail{0x380035f4}() # shadow
                                        # loadl 2/3
 17: t29 = DIRTY 1:I1 RdFX-gst(16,4) RdFX-gst(60,4)
::: helperc_LOADV32le{0x38006504}(t5) # shadow loadl
3/3
*18: t10 = LDle:I32(t5)               # *loadl
```

Memcheck: validity (aka, bitwise definedness)

When to flag use of uninitialized memory an error?

- C++ standard [13, sec 4.1, p1]: any lvalue-to-rvalue conversion on an uninitialized object has undefined behavior (C++14 relaxed for byte types).
- Valgrind works with bytes, padding causes a problem
    - eg, `struct Foo { char c, int32_t i };`
- Only operations which affect "observable" behavior are flagged as errors:
    - Control flow transfers
    - Conditional moves
    - Addresses used for memory access
    - Parameters to system calls

Memcheck: shadow memory

For each byte of application memory, keep track of:

- One ***A bit***: 0 means unaddressable, 1 means addressable.
- 8 ***V bits***: tracks bitwise definedness

=> need (up to) 257 bits per application bytes to track both.

Stored in something like a page table: address space is divided into 64K chunks. Each chunk is associated a "secondary map" (assume 32-bit address space):

```
typedef struct {
    uint8_t abits[8192];
    uint8_t vbits[65536];
} SM;
```

```
SM* PM[65536]; // "Primary map": covers 4GB
```

Instrumenting loads

```
U64 LOADVn(Addr a, SizeT nBits) {
  U1 abit; U8 vbits8; Int i;
  U64 vbits64 = V_BITS64_UNDEFINED;
  SizeT n_bad_addrs = 0;
  for (i = 0; i < nBits / 8; i++) {
    get_abit_and_vbits8(&abit, &vbits8, a + i);
    if (abit != A_BIT_ADDRESSABLE) {
      n_bad_addrs++;            // Defined-if
      vbits8 = V_BITS8_DEFINED; // unaddressable
    }
    vbits64 = (vbits64 << 8) | vbits8;
  }
  if (n_bad_addrs > 0)
    record_address_error(a, nBits);
  return vbits64;
}
```

Application loads are preceded by shadow loads.

Optimizations:

- Aligned loads can be done without a loop
- Vectorized range-setting
- Distinguished SMs: `NOACCESS, DEFINED, UNDEFINED`.
- **Compressed V-bits**: instead of 9 bits per byte, just use two, gives 4 states:
    - (1. Not addressable: `NOACCESS`), (2. addressable and completely defined: `DEFINED`), (3. addressable and completely undefined: `UNDEFINED`), (4. addressable and partially defined, `PARTDEFINED`)
- Faster stack updates.

Memcheck: malloc replacement

- Poisoned redzones flank heap-blocks
- Stacktraces written to heap-blocks in redzones
- Delays reuse of heap-blocks after free. Aka, "quarantine"

Memcheck: calculus of validity.

Let d1, d2 denote virtual registers, v1, v2 their corresponding shadow registers.

What are validity bits of, eg, d1 + d2?

Memcheck: calculus of validity.

Let d1, d2 denote virtual registers, v1, v2 their corresponding shadow registers.

What are validity bits of, eg, d1 + d2?

- `UifU(v1, v2)`? Undefined if either undefined?

Memcheck: calculus of validity.

Let d1, d2 denote virtual registers, v1, v2 their corresponding shadow registers.

What are validity bits of, eg, d1 + d2?

- `UifU(v1, v2)`? Undefined if either undefined?
- `Left(UifU(v1, v2))`? Overly pessimistic?

Memcheck

- Found innumerable bugs over the years
- Still useful.
- Slow. 20-40x slowdown for single-threaded performance

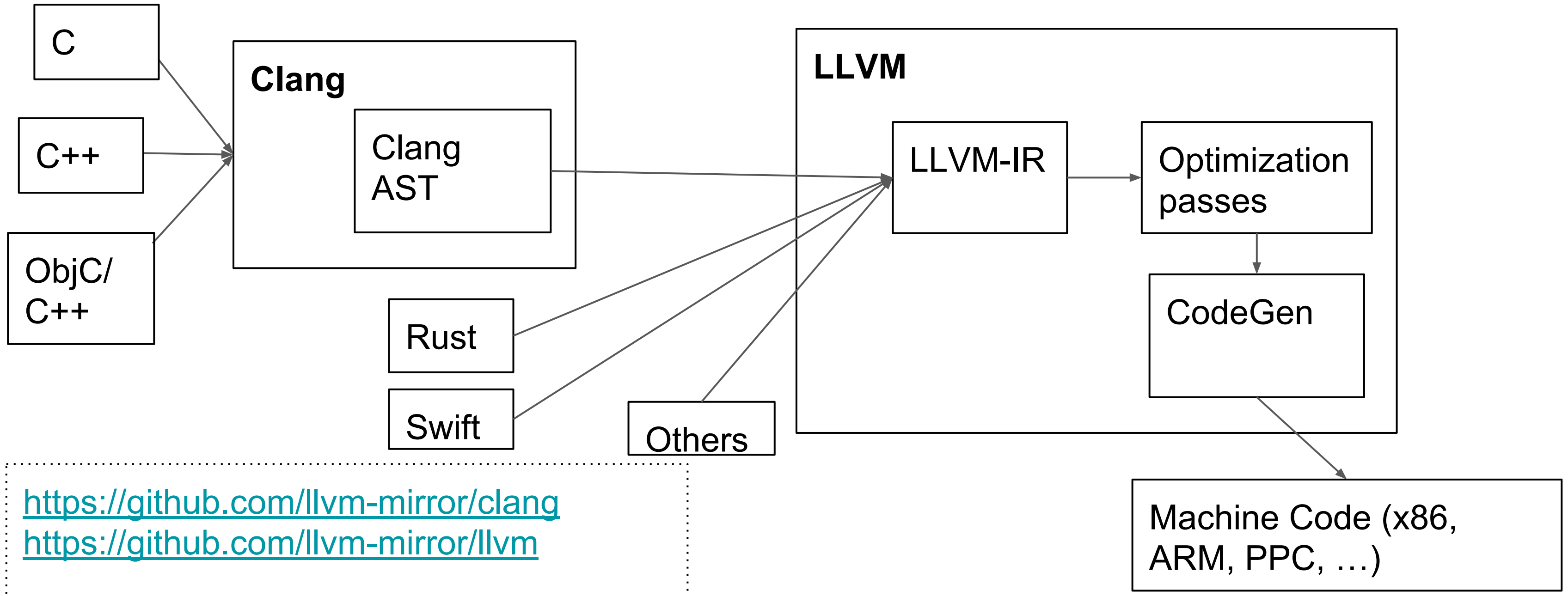
Google sanitizers

- AddressSanitizer (asan): addressability checks (A-bits)
- MemorySanitizer (msan, memsan): bitwise validity (definedness) (V-bits)
- ThreadSanitizer (tsan): data race detector
- UndefinedBehaviorSanitizer (ubsan): various "simple" checks, eg, integer overflow

All use compile-time instrumentation of code.

Kostya Serebryany

Quick tour of clang/llvm



https://github.com/llvm-mirror/clang
https://github.com/llvm-mirror/llvm

AddressSanitizer

- Out-of-bounds accesses to heap, stack and globals
- Use-after-free
- Stack use-after-return (runtime flag ASAN_OPTIONS=detect_stack_use_after_return=1)
- Use-after-scope (clang flag -fsanitize-address-use-after-scope)
- Double-free, invalid free
- Memory leaks (experimental)
- STL container over/underflow (!)

2-4x slowdown!

AddressSanitizer demo

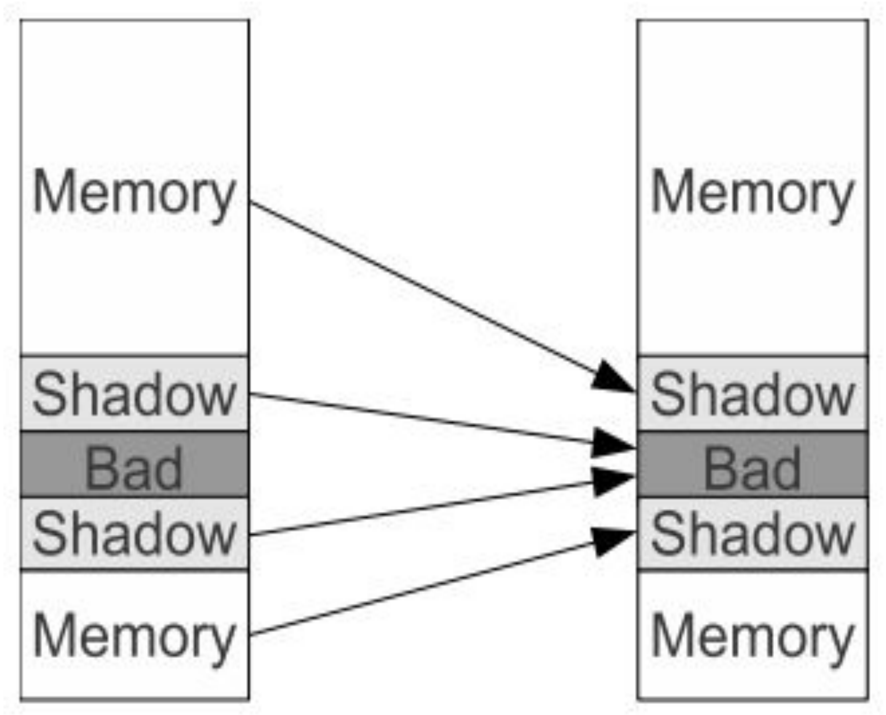
AddressSanitizer

Observation: addresses returned by malloc are typically (at least) 8-byte aligned (why?)

=> Each aligned 8-byte sequence of application heap memory is in one of 9 states: first k (0 <= k <= 8) bytes are addressable, remaining 8 - k bytes are not

=> Addressability of each aligned 8-byte sequence of application heap memory can be encoded by a single byte. 0 means completely addressible, k (0 <= k <= 8) means first k bytes are addressable, negatives distinguish between different unaddressable states (heap redzones, stack redzones, global redzones, …)

Asan Shadow mapping: `shadow(Addr) = (Addr >> 3) + Offset;`



32-bit systems:

`Offset` = 2**29

64-bit systems:

`Offset` = 2 ** 44

Asan Instrumentation

```
// 8 byte loads
ShadowAddr = (Addr >> 3) + Offset;
if (*ShadowAddr != 0)
    ReportAndCrash(Addr);

// 1-,2-, or 4-byte accesses
ShadowAddr = (Addr >> 3) + Offset;
k = *ShadowAddr;
if (k != 0 && ((Addr & 7) + AccessSize) > k)
    ReportAndCrash(Addr);
```

Implemented in
https://github.com/llvm-mirror/llvm/blob/master/lib/Transforms/Instrumentation/AddressSanitizer.cpp

Asan Runtime Library

https://github.com/llvm-mirror/compiler-rt/tree/master/lib/asan

libasan.{so|dylib}

- Malloc replacement
  - Redzone insertion
  - Free quarantine
- Global redzone initialization
- Error reporting

Asan, memcheck comparison

Asan wins:

- **Speed**
- Instrumentation of globals, better stack instrumentation, use after return, container overflow

Memcheck wins:

- Also checks (bitwise) validity
- Inline assembly, jitted code, external libs handled
- Requires no compilation, linking changes.

MemorySanitizer

Detector of uninitialized memory reads (UMR) in C/C++.

Similar to validity checking in valgrind.

(bitwise) Validity calculus is less precise than Valgrind

https://github.com/llvm-mirror/llvm/blob/master/lib/Transforms/Instrumentation/MemorySanitizer.cpp

https://github.com/llvm-mirror/compiler-rt/tree/master/lib/msan

Memsan example

```
% cat umr2.cc
#include <stdio.h>

int main(int argc, char** argv) {
  int* a = new int[10];
  a[5] = 0;
  volatile int b = a[argc];
  if (b)
    printf("xx\n");
  return 0;
}

% clang -fsanitize=memory \
-fsanitize-memory-track-origins=2 \
-fno-omit-frame-pointer -g -O2 \
umr2.cc
```

```
% ./a.out
WARNING: MemorySanitizer: use-of-uninitialized-value
    #0 0x7f7893912f0b in main umr2.cc:7
    #1 0x7f789249b76c in __libc_start_main
libc-start.c:226

  Uninitialized value was stored to memory at
    #0 0x7f78938b5c25 in __msan_chain_origin msan.cc:484
    #1 0x7f7893912ecd in main umr2.cc:6

  Uninitialized value was created by a heap allocation
    #0 0x7f7893901cbd in operator new[](unsigned long)
msan_new_delete.cc:44
    #1 0x7f7893912e06 in main umr2.cc:4
```

Memsan, memcheck comparison

Memsan wins:

- Speed
- Better origin tracking, knows variable names

Memcheck wins:

- Also checks addressability
- Inline assembly, jitted code, shared libs all OK
- **Doesn't require all libs to be instrumented**

ThreadSanitizer

Data race detector based on compile-time instrumentation.

Slowdown 5x-15x

Memory overhead 5x-10x

Tsan demo

Tsan shadow mapping

Shadow State is N Shadow Words (described below); N is one of 2, 4, 8 (configurable). Every aligned 8-byte word of application memory is mapped into N Shadow Words using direct address mapping.

Shadow Word, a 64-bit object:

TID (Thread Id):	16 bits (configurable)
Scalar Clock:	42 bits (configurable)
IsWrite:	1 bit
Access Size (1, 2, 4 or 8):	2 bits
Address Offset (0..7):	3 bits

One Shadow Word represents a single memory access to a subset of bytes within the 8-byte word of application memory. Therefore the Shadow State describes N different accesses to the corresponding application memory region.

UndefinedBehaviorSanitizer (1)

`-fsanitize=alignment`: Use of a misaligned pointer or creation of a misaligned reference.
`-fsanitize=bool`: Load of a `bool` value which is neither `true` nor `false`.
`-fsanitize=builtin`: Passing invalid values to compiler builtins.
`-fsanitize=bounds`: Out of bounds array indexing, in cases where the array bound can be statically determined.
`-fsanitize=enum`: Load of a value of an enumerated type which is not in the range of representable values for that enumerated type.
`-fsanitize=float-cast-overflow`: Conversion to, from, or between floating-point types which would overflow the destination.
`-fsanitize=float-divide-by-zero`: Floating point division by zero.
`-fsanitize=function`: Indirect call of a function through a function pointer of the wrong type (Darwin/Linux, C++ and x86/x86_64 only).
`-fsanitize=implicit-integer-truncation`: Implicit conversion from integer of larger bit width to smaller bit width, if that results in data loss. That is, if the demoted value, after casting back to the original width, is not equal to the original value before the downcast. Issues caught by this sanitizer are not undefined behavior, but are often unintentional.
`-fsanitize=integer-divide-by-zero`: Integer division by zero.
`-fsanitize=nonnull-attribute`: Passing null pointer as a function parameter which is declared to never be null.
`-fsanitize=null`: Use of a null pointer or creation of a null reference.
`-fsanitize=nullability-arg`: Passing null as a function parameter which is annotated with `_Nonnull`.
`-fsanitize=nullability-assign`: Assigning null to an lvalue which is annotated with `_Nonnull`.

Ubsan (2)

-fsanitize=nullability-return: Returning null from a function with a return type annotated with _Nonnull.
-fsanitize=object-size: An attempt to potentially use bytes which the optimizer can determine are not part of the object being accessed. This will also detect some types of undefined behavior that may not directly access memory, but are provably incorrect given the size of the objects involved, such as invalid downcasts and calling methods on invalid pointers. These checks are made in terms of __builtin_object_size, and consequently may be able to detect more problems at higher optimization levels.
-fsanitize=pointer-overflow: Performing pointer arithmetic which overflows.
-fsanitize=return: In C++, reaching the end of a value-returning function without returning a value.
-fsanitize=returns-nonnull-attribute: Returning null pointer from a function which is declared to never return null.
-fsanitize=shift: Shift operators where the amount shifted is greater or equal to the promoted bit-width of the left hand side or less than zero, or where the left hand side is negative. For a signed left shift, also checks for signed overflow in C, and for unsigned overflow in C++. You can use -fsanitize=shift-base or -fsanitize=shift-exponent to check only left-hand side or right-hand side of shift operation, respectively.
-fsanitize=signed-integer-overflow: Signed integer overflow, where the result of a signed integer computation cannot be represented in its type. This includes all the checks covered by -ftrapv, as well as checks for signed division overflow (INT_MIN/-1), but not checks for lossy implicit conversions performed before the computation (see -fsanitize=implicit-conversion). Both of these two issues are handled by -fsanitize=implicit-conversion group of checks.
-fsanitize=unreachable: If control flow reaches an unreachable program point.

Ubsan (3)

`-fsanitize=unsigned-integer-overflow`: Unsigned integer overflow, where the result of an unsigned integer computation cannot be represented in its type. Unlike signed integer overflow, this is not undefined behavior, but it is often unintentional. This sanitizer does not check for lossy implicit conversions performed before such a computation (see `-fsanitize=implicit-conversion`).

`-fsanitize=vla-bound`: A variable-length array whose bound does not evaluate to a positive value.

`-fsanitize=vptr`: Use of an object whose vptr indicates that it is of the wrong dynamic type, or that its lifetime has not begun or has ended. Incompatible with `-fno-rtti`. Link must be performed by `clang++`, not `clang`, to make sure C++-specific parts of the runtime library and C++ standard libraries are present.

Ubsan demo

Practical recommendations

Use these tools. (and static analysis too)

How should you take advantage of these tools?

- Run them as necessary, much like debuggers.
- Run them on your unit tests, eg nightly or weekly memory checks.
  - This is particularly easy for valgrind and cmake: `ctest -T memcheck`
  - For sanitizers, it requires multiple rebuilds (no two sanitizers which require shadow memory can be run simultaneously)
- Run instrumented fuzz tests.

Csmith: generator of valid c99 programs

```
// an example csmith program which
// at some point crashed clang

typedef signed char int8_t;
typedef short int int16_t;
typedef int int32_t;
typedef unsigned char uint8_t;
typedef unsigned int uint32_t;
uint32_t g_2 = 8L;
volatile int16_t g_133 = 0x7EAAL;
uint32_t g_139 = 0L;
volatile uint8_t g_160 = 6L;
int32_t func_31 (int16_t p_33,
                 int16_t p_34,
                 int32_t p_35);
int32_t func_97 (uint32_t p_98);
```

```
int32_t func_97 (uint32_t p_98)
{
  uint8_t l_190 = 0xFAL;
  int8_t l_163 = 1L;
  for (p_98 = -26; ((g_160) >= -29); (g_160)--)
    {
      int8_t l_192 = 0x85L;
      func_31 (((((l_190 / 1L) - p_98) ^ ((g_139 + p_98) ^
0xA2D7L))
            && g_2), (((((g_133) && l_192) || p_98) | l_163) >
1L), p_98);
    }
}
```

Final points

- Dynamic analysis and fuzzing have found thousands of bugs in Chrome
- Despite this, "canaries" still find bugs that elude Google's best efforts
- Most common addressability bug at Google: **use after free**, followed by **heap buffer overflow**.

References

Valgrind

- http://valgrind.org/
- Valgrind: a Framework for Heavyweight Dynamic Binary Instrumentation (http://valgrind.org/docs/valgrind2007.pdf)
- Using Valgrind to detect undefined value errors with bit-precision (http://valgrind.org/docs/memcheck2005.pdf)
- How to shadow every byte of memory used by a program (http://www-leland.stanford.edu/class/cs343/resources/shadow-memory2007.pdf)

AddressSanitizer

- https://clang.llvm.org/docs/AddressSanitizer.html
- https://www.usenix.org/system/files/conference/atc12/atc12-final39.pdf

MemorySanitizer

- https://clang.llvm.org/docs/MemorySanitizer.html
- https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/43308.pdf

ThreadSanitizer

- https://clang.llvm.org/docs/ThreadSanitizer.html
- ThreadSanitizer – data race detection in practice (https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/35604.pdf). *Note: dated -- describes an old TSAN*

Fuzzing

- https://llvm.org/docs/LibFuzzer.html
- http://lcamtuf.coredump.cx/afl/