COMPSCI2030 Systems Programming

Sanitizers

Yehia Elkhatib

VIA VERITAS VITA

University of Glasgow

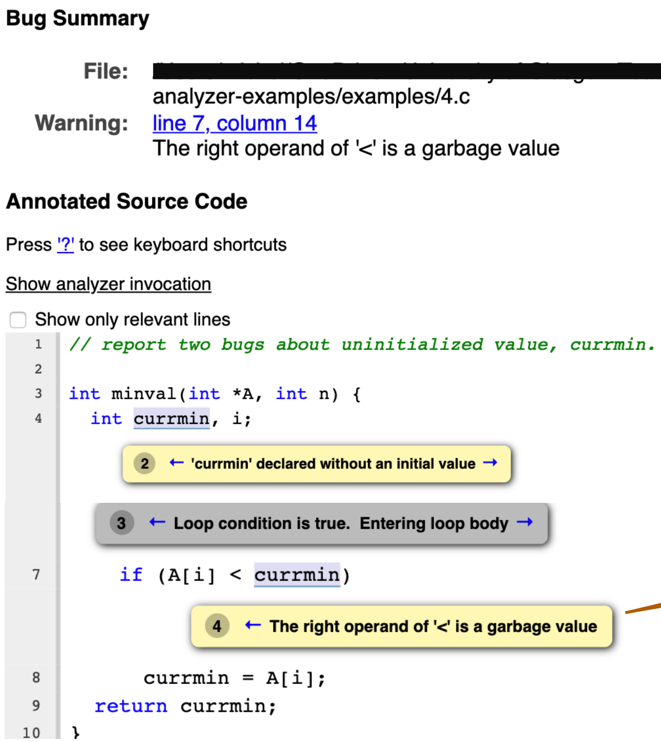
Static Analysis

- A static analyzer reasons about the code *without* executing it
- The compiler performs some static analysis every time you compile code
  - e.g. type checking
- It is good practice to enable all warnings and make warnings errors
  - i.e. use the flags `-Wall -Werror`
  - so the compiler can be most helpful in detecting bugs before execution
- Some static analysis is too expensive to perform in every build
  Other static analysis enforces a particular coding guideline
- We invoke the static analyzer using a flag and specifying the output format of the report:

```
clang --analyze --analyzer-output html program.c
```

clang static analyzer report

- The tool generates a report explaining potential bugs



Bug Summary

File: [redacted]/slides/SystemsProgramming/lecture5examples/clang-analyzer-examples/examples/4.c

Warning: line 7, column 14
The right operand of '<' is a garbage value

Annotated Source Code

Press '?' to see keyboard shortcuts

Show analyzer invocation

□ Show only relevant lines

```
1  // report two bugs about uninitialized value, currmin.
2
3  int minval(int *A, int n) {
4    int currmin, i;
         2 ← 'currmin' declared without an initial value →
       3 ← Loop condition is true.  Entering loop body →
7      if (A[i] < currmin)
             4 ← The right operand of '<' is a garbage value
8        currmin = A[i];
9    return currmin;
10 }
```

Here the tool has reported a “garbage value” which in this case leads to undefined outcome for an if-branch

clang-tidy

- A *linter* (the name comes from the first UNIX tool to perform static analysis on C)
- `clang-tidy` is a flexible tool that allows to enforce coding guidelines and to modernize source code
  - It is possible to extend `clang-tidy` by writing your own checks
- It is invoked like `clang`, accepting the same flags but after two dashes: `--`
- A series of *checks* can be enabled / disabled

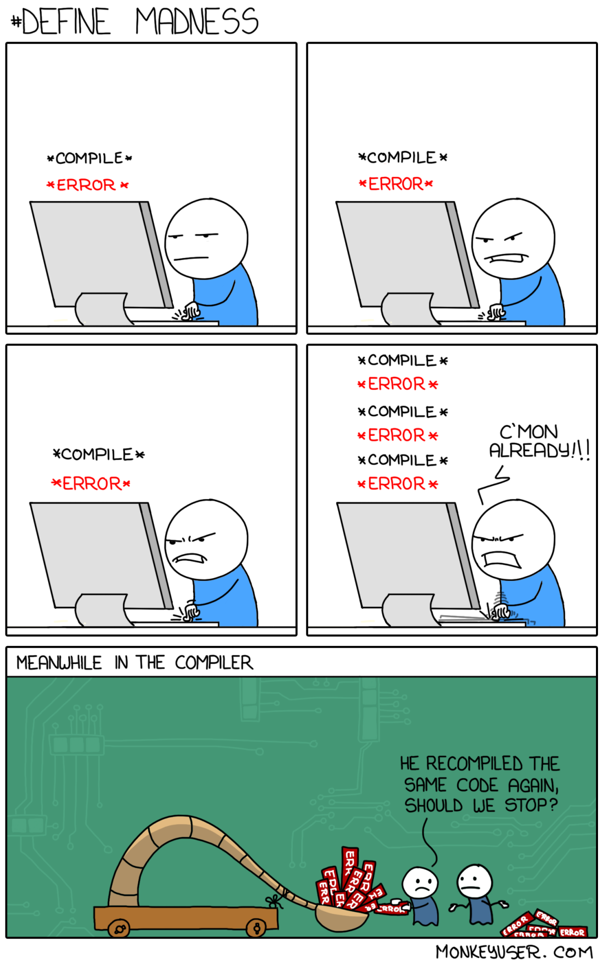
```
$ clang-tidy -checks="readability-*" 6.c -- -Wall -Werror
/Users/lito/Desktop/examples/6.c:2:16: warning: pointer parameter 'p' can be pointer to const
[readability-non-const-parameter]
void test(int *p) {
           ~~~ ^
          const
/Users/lito/Desktop/examples/6.c:4:9: warning: statement should be inside braces
[readability-braces-around-statements]
if (p)
      ^
       {
```

Here we enable the checks for *readability*

It suggests to to use const and put braces around the branch of an if statement

- Detailed information is available at: http://clang.llvm.org/extra/clang-tidy/

Use the analyzer to improve your code



Dynamic Analysis Tools

- There exists a family of bug detection tools that use dynamic analysis
- These tools need the program to run and can only detect bugs which are encountered during the execution of a particular test input
- The `clang` project calls these tools *sanitizers*. The most important are:
  - `AddressSanitizer` - a memory error detector
  - `MemorySanitizer` - a detector of uninitialized reads
  - `LeakSanitizer` - a memory leak detector
  - `UndefinedBehaviorSanitizer` - a detector of undefined behaviour
- Later in the course, you might want to look up:
  - `ThreadSanitizer` - a data race detector

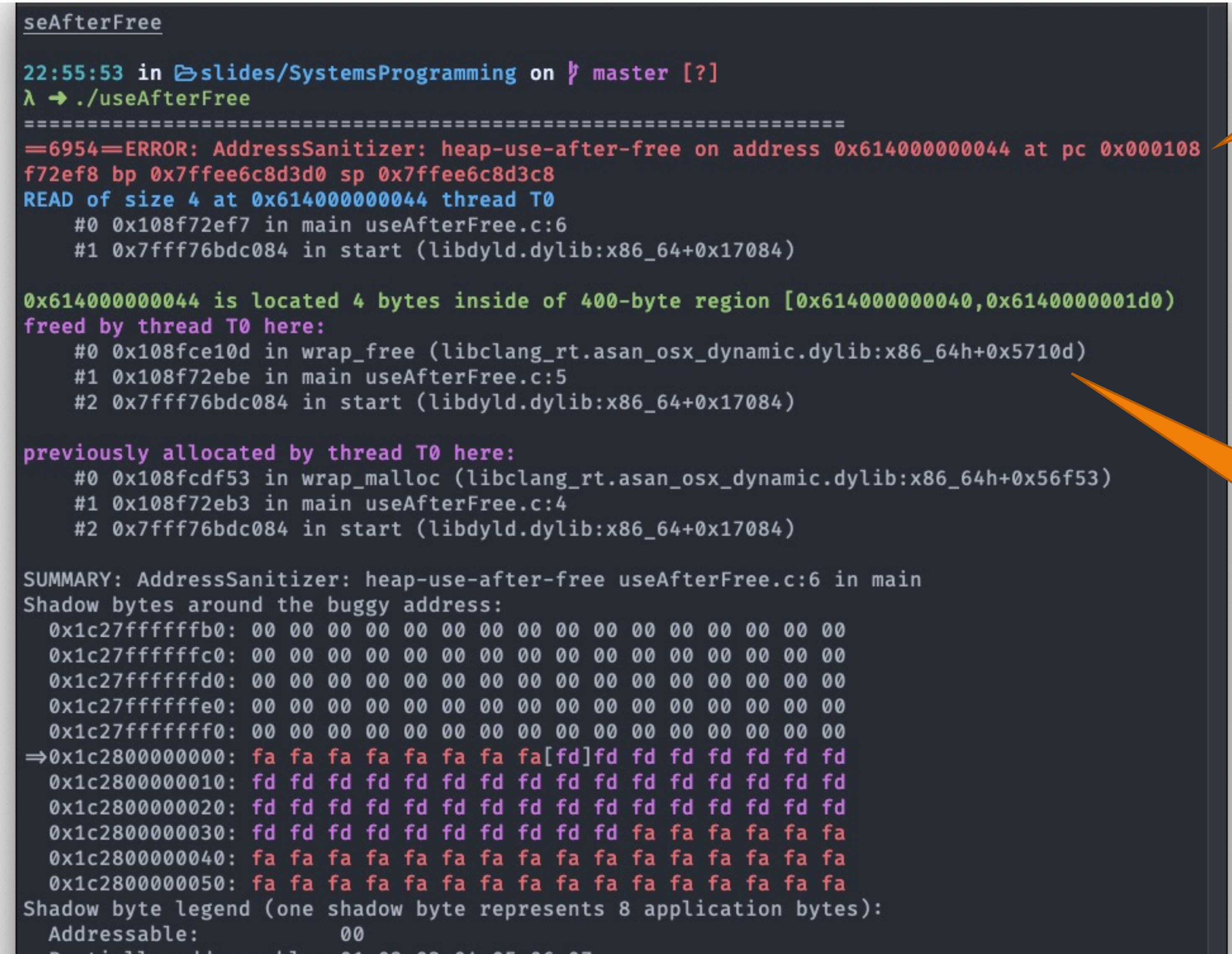
Address Sanitizer

- Address Sanitizer is a memory error detector for:
  - Out-of-bounds / Use-after-free / Double free memory accesses
- Makes clang insert instructions to monitor every memory access
- This slows down the execution by about 2x
  - `valgrind` (a similar tool) has often a slowdown of 20-100x!
- These flags enable address sanitizer:

```
clang -fsanitize=address -fno-omit-frame-pointer -O1 -g -Wall -Werror program.c -o program
```

  - `fno-omit-frame-pointer` produces a readable call stack
  - `O1` enables basic optimizations
- The compiler will produce a binary as usual: `./program`
- Address Sanitizer has found hundreds of bugs in large scale software projects
  - e.g. Chromium and Firefox

Address Sanitizer output



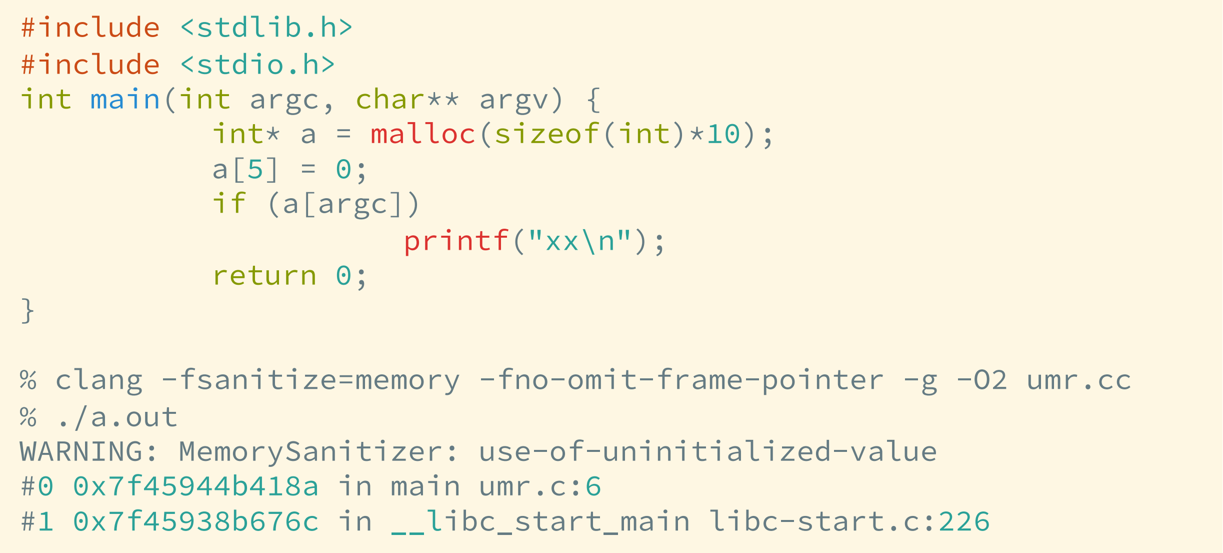
```
seAfterFree

22:55:53 in 🗁slides/SystemsProgramming on ᚴ master [?]
λ ➜ ./useAfterFree
=================================================================
==6954==ERROR: AddressSanitizer: heap-use-after-free on address 0x614000000044 at pc 0x000108
f72ef8 bp 0x7ffee6c8d3d0 sp 0x7ffee6c8d3c8
READ of size 4 at 0x614000000044 thread T0
    #0 0x108f72ef7 in main useAfterFree.c:6
    #1 0x7fff76bdc084 in start (libdyld.dylib:x86_64+0x17084)

0x614000000044 is located 4 bytes inside of 400-byte region [0x614000000040,0x6140000001d0)
freed by thread T0 here:
    #0 0x108fce10d in wrap_free (libclang_rt.asan_osx_dynamic.dylib:x86_64h+0x5710d)
    #1 0x108f72ebe in main useAfterFree.c:5
    #2 0x7fff76bdc084 in start (libdyld.dylib:x86_64+0x17084)

previously allocated by thread T0 here:
    #0 0x108fcdf53 in wrap_malloc (libclang_rt.asan_osx_dynamic.dylib:x86_64h+0x56f53)
    #1 0x108f72eb3 in main useAfterFree.c:4
    #2 0x7fff76bdc084 in start (libdyld.dylib:x86_64+0x17084)

SUMMARY: AddressSanitizer: heap-use-after-free useAfterFree.c:6 in main
Shadow bytes around the buggy address:
  0x1c27ffffffb0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x1c27ffffffc0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x1c27ffffffd0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x1c27ffffffe0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x1c27fffffff0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
=>0x1c2800000000: fa fa fa fa fa fa fa fa[fd]fd fd fd fd fd fd fd
  0x1c2800000010: fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd
  0x1c2800000020: fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd
  0x1c2800000030: fd fd fd fd fd fd fd fd fd fd fa fa fa fa fa fa
  0x1c2800000040: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x1c2800000050: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
```

Reports a heap-use-after-free on address 0x614000000044

Provides information where the memory was freed (line 5) and allocated (line 4)

Memory Sanitizer

- Detects uninitialized reads to memory

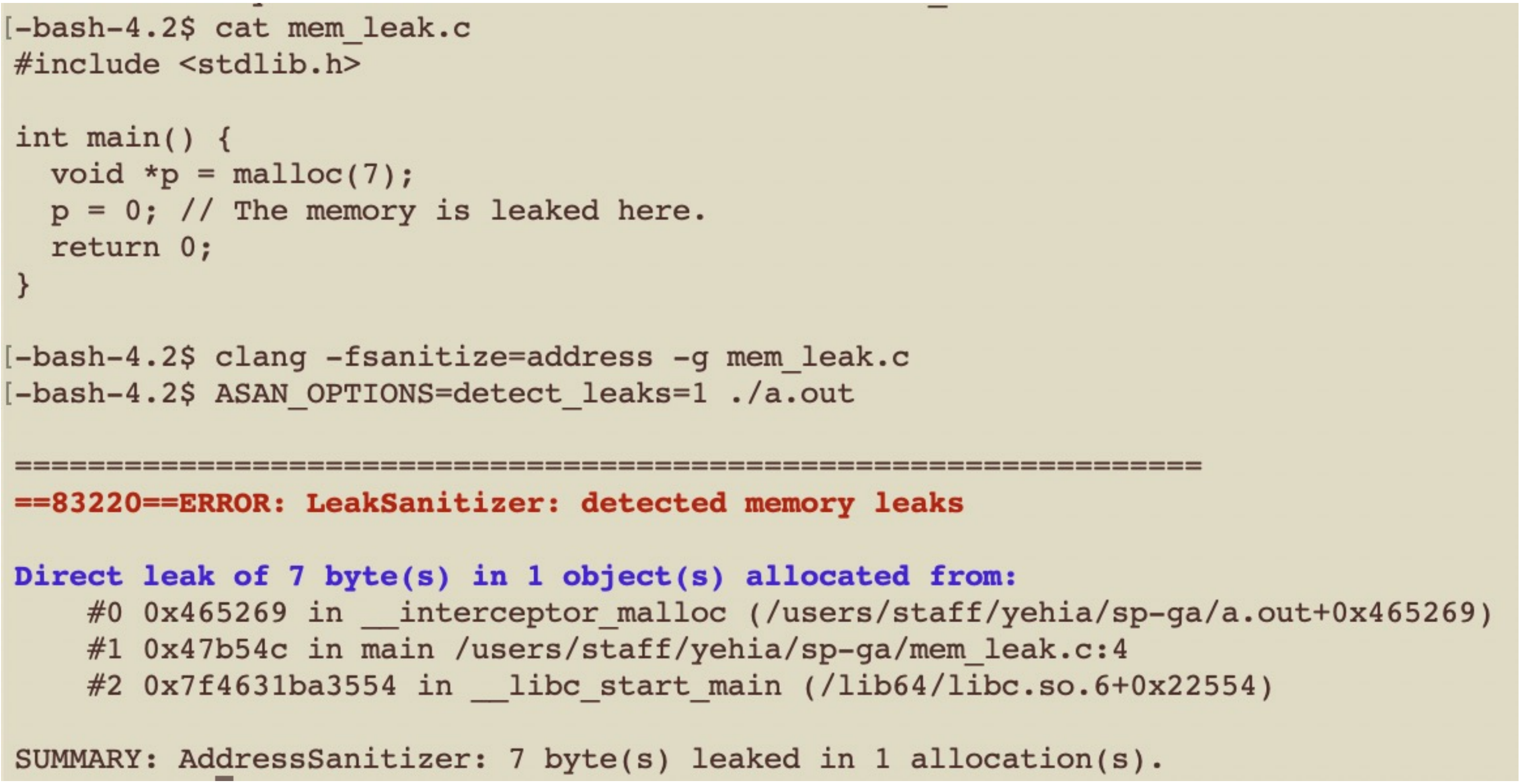


```
#include <stdlib.h>
#include <stdio.h>
int main(int argc, char** argv) {
           int* a = malloc(sizeof(int)*10);
           a[5] = 0;
           if (a[argc])
                      printf("xx\n");
           return 0;
}

% clang -fsanitize=memory -fno-omit-frame-pointer -g -O2 umr.cc
% ./a.out
WARNING: MemorySanitizer: use-of-uninitialized-value
#0 0x7f45944b418a in main umr.c:6
#1 0x7f45938b676c in __libc_start_main libc-start.c:226
```

- Under active development
- Currently only available for Linux and BSD

Leak Sanitizer

- Leak sanitizer detects memory leaks (i.e. un-free’d memory blocks)



```
[-bash-4.2$ cat mem_leak.c
#include <stdlib.h>

int main() {
  void *p = malloc(7);
  p = 0; // The memory is leaked here.
  return 0;
}

[-bash-4.2$ clang -fsanitize=address -g mem_leak.c
[-bash-4.2$ ASAN_OPTIONS=detect_leaks=1 ./a.out

==================================================================
==83220==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 7 byte(s) in 1 object(s) allocated from:
    #0 0x465269 in __interceptor_malloc (/users/staff/yehia/sp-ga/a.out+0x465269)
    #1 0x47b54c in main /users/staff/yehia/sp-ga/mem_leak.c:4
    #2 0x7f4631ba3554 in __libc_start_main (/lib64/libc.so.6+0x22554)

SUMMARY: AddressSanitizer: 7 byte(s) leaked in 1 allocation(s).
```

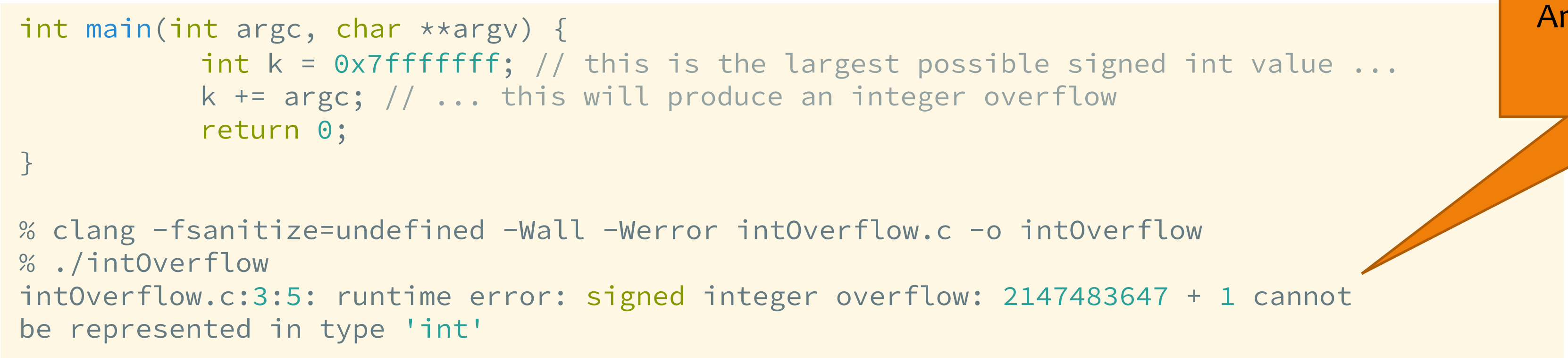
- Under active development
- Available for Linux, macOS, NetBSD

Undefined Behaviour

- For certain operations, the C standard demands no particular behaviour
  - typically cases that are considered bugs, e.g. dereferencing a `null` pointer
- It is expensive to check if dereferencing a pointer is valid <u>every time</u>
- Because the compiler does not have to ensure a particular behaviour for `null` pointers, it *assumes* that the programmer checked that the pointer is not `null`
- Undefined behaviour is therefore crucial for fast code, but makes detection of bugs much harder, as it is not guaranteed that a bug will result in a crash
- A good introduction to undefined behaviour is this series of blog posts: What Every C Programmer Should Know About Undefined Behavior

Undefined Behaviour Sanitizer

- Detects various types of undefined behaviour



```
int main(int argc, char **argv) {
          int k = 0x7fffffff; // this is the largest possible signed int value ...
          k += argc; // ... this will produce an integer overflow
          return 0;
}

% clang -fsanitize=undefined -Wall -Werror intOverflow.c -o intOverflow
% ./intOverflow
intOverflow.c:3:5: runtime error: signed integer overflow: 2147483647 + 1 cannot
be represented in type 'int'
```

An integer overflow is detected