Programming in C and C++

Lecture 5: Tooling

Neel Krishnaswami and Alan Mycroft

Undefined and Unspecified Behaviour

- We have seen that C is an *unsafe* language
- Programming errors can arbitrarily corrupt runtime data structures...
- ...leading to *undefined behaviour*
- Enormous number of possible sources of undefined behavior (See `https://blog.regehr.org/archives/1520`)
- What can we do about it?

Tooling and Instrumentation

Add instrumentation to detect unsafe behaviour!

We will look at 4 tools:

- ASan (Address Sanitizer)
- MSan (Memory Sanitizer)
- UBSan (Undefined Behaviour Sanitizer)
- Valgrind

ASan: Address Sanitizer

- One of the leading causes of errors in C is memory corruption:
  - Out-of-bounds array accesses
  - Use pointer after call to `free()`
  - Use stack variable after it is out of scope
  - Double-frees or other invalid frees
  - Memory leaks
- AddressSanitizer instruments code to detect these errors
- Need to recompile
- Adds runtime overhead
- Use it while developing
- Built into gcc and clang!

ASan Example #1

```c
#include <stdlib.h>
#include <stdio.h>

#define N 10

int main(void) {
  char s[N]  = "123456789";
  for (int i = 0; i <= N; i++)
    printf ("%c", s[i]);
  printf("\n");
  return 0;
}
```

- Loop bound goes past the end of the array
- Undefined behaviour!
- Compile with `-fsanitize=address`

ASan Example #2

```c
#include <stdlib.h>

int main(void) {
  int *a =
    malloc(sizeof(int) * 100);
  free(a);
  return a[5]; // DOOM!
}
```

1. array is allocated
2. array is freed
3. array is dereferenced! (aka use-after-free)

ASan Example #3

```c
#include <stdlib.h>

int main(void) {
  char *s =
    malloc(sizeof(char) * 10);
  free(s);
  free(s);
  printf("%s", s);
  return 0;
}
```

1. array is allocated
2. array is freed
3. array is double-freed

ASan Limitations

- Must recompile code
- Adds considerable runtime overhead
  - Typical slowdown 2x
- Does not catch all memory errors
  - NEVER catches *uninitialized* memory accesses
- Still: a **must-use** tool during development

MSan: Memory Sanitizer

- Both local variable declarations and dynamic memory allocation via `malloc()` do not initialize memory:

```c
#include <stdio.h>

int main(void) {
  int x[10];
  printf("%d\n", x[0]); // uninitialized
  return 0;
}
```

- Accesses to uninitialized variables are undefined
  - This does *NOT* mean that you get some unspecified value
  - It means that the compiler is free to do *anything it likes*
- ASan does not catch *uninitialized memory accesses*

MSan: Memory Sanitizer

```c
#include <stdio.h>

int main(void) {
  int x[10];
  printf("%d\n", x[0]); // uninitialized
  return 0;
}
```

- Memory sanitizer (MSan) does check for uninitialized memory accesses
- Compile with `-fsanitize=memory`

MSan Example #1: Stack Allocation

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char** argv) {
5    int a[10];
6    a[2] = 0;
7    if (a[argc])
8      printf("print something\n");
9    return 0;
10 }
```

1. Stack allocate array on line 5
2. Partially initialize it on line 6
3. Access it on line 7
4. This might or might not be initialized

MSan Example #2: Heap Allocation

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char** argv) {
5    int *a = malloc(sizeof(int) * 10);
6    a[2] = 0;
7    if (a[argc])
8      printf("print something\n");
9    free(a);
10   return 0;
11 }
```

1. Heap allocate array on line 5
2. Partially initialize it on line 6
3. Access it on line 7
4. This might or might not be initialized

MSan Limitations

- MSan just checks for memory initialization errors
- It is very expensive
  - 2-3x slowdowns, on top of anything else
- Currently only available on clang, and not gcc

UBSan: Undefined Behaviour Sanitizer

- There is lots of non-memory-related undefined behaviour in C:
  - Signed integer overflow
  - Dereferencing null pointers
  - Pointer arithmetic overflow
  - Dynamic arrays whose size is non-positive
- Undefined Behaviour Sanitizer (UBSan) instruments code to detect these errors
- Need to recompile
- Adds runtime overhead
  - Typical overhead of 20%
- Use it while developing, maybe even in production
- Built into gcc and clang!

UBSan Example #1

```c
#include <limits.h>

int main(void) {
  int n = INT_MAX;
  int m = n + 1;
  return 0;
}
```

1. Signed integer overflow is undefined
2. So value of `m` is undefined
3. Compile with `-fsanitize=undefined`

UBSan Example #2

```
1  #include <limits.h>
2
3  int main(void) {
4    int n = 65
5    int m = n / (n - n);
6    return 0;
7  }
```

1. Division-by-zero is undefined
2. So value of `m` is undefined
3. Any possible behaviour is legal!

UBSan Example #3

```
1  #include <stdlib.h>
2
3  struct foo {
4    int a, b;
5  };
6
7  int main(void) {
8    struct foo *x = NULL;
9    int m = x->a;
10   return 0;
11 }
```

1. Accessing a null pointer is undefined
2. So accessing fields of `x` is undefined
3. Any possible behaviour is legal!

UBSan Limitations

- Must recompile code
- Adds modest runtime overhead
- Does not catch all undefined behaviour
- Still: a **must-use** tool during development
- **Seriously consider** using it in production

Valgrind

- UBSan, MSan, and ASan require recompiling
- UBSan and ASan don't catch accesses to uninitialized memory
- Enter *Valgrind*!
- Instruments binaries to detect numerous errors

Valgrind Example

```c
#include <stdio.h>

int main(void) {
  char s[10];
  for (int i = 0; i < 10; i++)
    printf("%c", s[i]);
  printf("\n");
  return 0;
}
```

1. Accessing elements of `s` is undefined
2. Program prints uninitialized memory
3. Any possible behaviour is legal!
4. Invoke `valgrind` with binary name

Valgrind Limitations

- Adds very substantial runtime overhead
- Not built into GCC/clang (plus or minus?)
- As usual, does not catch all undefined behaviour
- Still: a **must-use** tool during testing

Summary

Tool	Slowdown	Source/Binary	Tool
ASan	Big	Source	GCC/Clang
MSan	Big	Source	Clang
UBSan	Small	Source	GCC/Clang
Valgrind	Very big	Binary	Standalone