



Undefined Behaviour in C

Zhenpeng Wu, Zixuan Yin, Lucas Zamprogno



With undefined behaviour, anything is possible!



Overview

In this presentation we will discuss

1. Categories of behaviour in C
2. How undefined behaviour in C works
3. Examples and instances of undefined behaviour
4. Mitigation strategies and research on undefined behaviour for C



Learning Objectives

By the end of this presentation you should be able to:

- List and describe different categories of behaviour in C
- Describe the pros and cons of undefined behaviour
- Give an example of undefined behaviour in C
- Give an example of how risks from undefined behaviour can be mitigated



C Background

- Originally developed 40+ years ago
- Widely used for systems programming
- An efficient but unsafe low level programming language - errors are not trapped
- Mainstream C compilers include Visual C++, GNU Compiler Collection (GCC), and Clang/Low Level Virtual Machine (LLVM)
- Compilers compile code based on the standard; can perform any code transformations (e.g. reordering, deleting) that do not change the observable behavior of the program



Philosophy


TRUST THE PROGRAMMER





Types of Behaviour in C: Unspecified

- Behaviour where the standard provides two or more possibilities and imposes no further requirements on which is chosen in any instance
- May produce an executable that exhibits different behavior when compiled on a different compiler, or on the same compiler with different settings, or in different parts of the same executable.
- Examples:
 - Order of evaluation of the operands of any C operator (e.g. function arguments, subexpressions)
 - Whether identical string literals are distinct



Types of Behaviour in C: Implementation Defined

- Unspecified behavior where each implementation documents how the choice is made
- Portable programs should try to avoid such behaviour
- Examples:
 - Number of bits in a byte
 - Whether signed integer right shift is arithmetic or logical.



Types of Behaviour in C: Undefined

- The C standard does not put any restrictions on the behavior of the program
- Examples:
 - Out of bound array access
 - Signed integer overflow
 - Null pointer dereference
 - Access to an object through a pointer of a different type
 - ... C99 shows 195 forms of undefined behavior



Types of Behaviour in C: Undefined

- Compilers are not required to diagnose undefined behavior and are free to do anything
 - Deleting your hard drive is technically valid
 - Devs wouldn't like your compiler much
- Undefined behaviour makes the entire program's behaviour undefined
 - Not just the statement
 - Not just what's after the statement



Why would you want this?

- Simplifies the compiler's job, allowing it to generate very efficient code in certain situations
 - Also, doesn't have to generate bad code to force same behaviours on different architectures
 - x86 errors on divide by 0, PowerPC ignores it
- These situations often involve tight loops
 - No need to perform bound checks for array accesses
 - No need to check whether the counter overflows



Why wouldn't you want this?

- Programmers need to know about undefined behaviour and remember to avoid them
- The consequence of undefined behavior is unpredictable
 - Run correctly
 - Crash - which might also be good
 - Silently generate incorrect results
 - Run on one machine, but crash on another
- Make the entire program meaningless
- Security vulnerabilities (e.g. stack smashing attack)



Can't compilers help with this?

- Can provide warnings for simple cases (optional flags)
- Create their own definitions for behaviour
- Provide sensible implementations

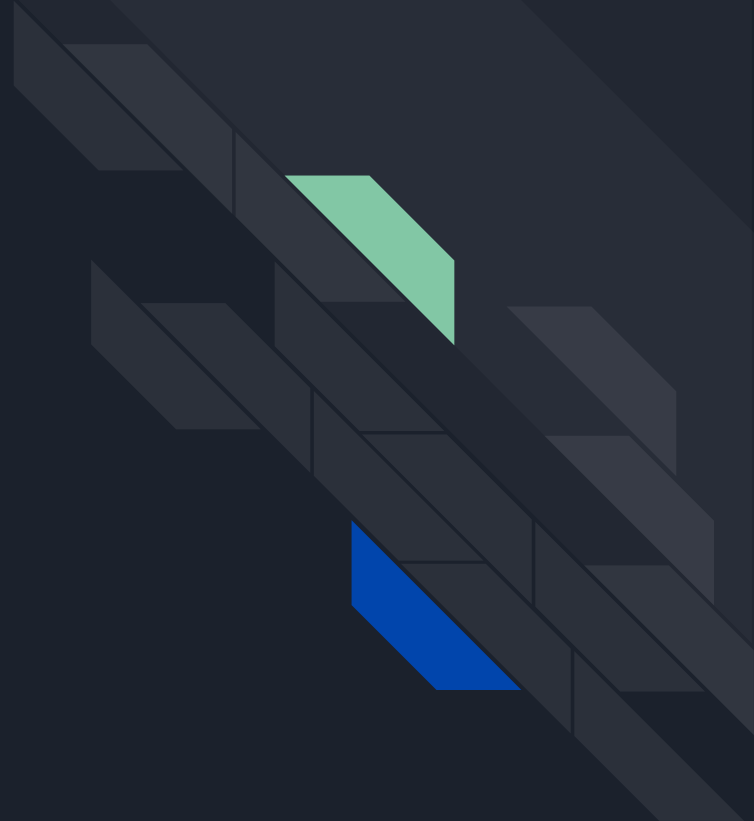


Can't compilers help with this?

- Could come at a speed cost
- Programmer reliance on compiler specification is dangerous
 - Might not think when changing compilers
 - Compiler updates could have breaking changes
- Identifying dangerous undefined behaviours after multiple optimizations (and explaining them) is challenging or impossible

Cases and Examples

With some audience participation :D





Use of an Uninitialized Variable

Note: Doesn't apply to static and global variables

```
int main() {  
    int i, counter;  
    for(i = 0; i < 10; ++i)  
        counter += i;  
    printf("%d\n", counter);  
    return 0;  
}
```




Use of an Uninitialized Variable

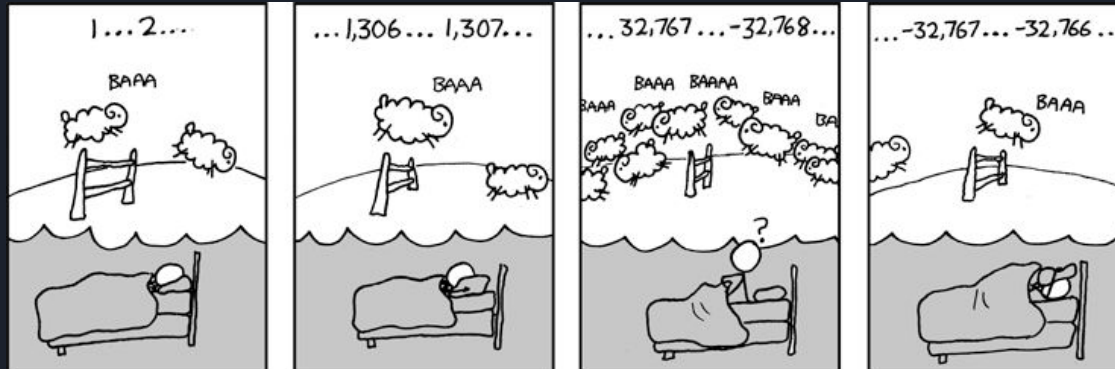
- Possible behaviour:
Keep value at
memory address
- Faster when
allocating larger
amounts of memory

```
int main() {  
    int i, counter;  
    for(i = 0; i < 10; ++i)  
        counter += i;  
    printf("%d\n", counter);  
    return 0;  
}
```

Signed Integer Overflow

```
int main() {  
    for (i = 0; i <= N; ++i) { ... }  
}
```

Behaviour is undefined when $i = N = \text{INT_MAX}$





Signed Integer Overflow

```
int main() {  
    for (i = 0; i <= N; ++i) { ... }  
}
```

- Possible behaviour: $\text{INT_MAX} + 1 == \text{INT_MIN}$
- Compiler doesn't have to consider that this could run forever
- Allows optimizations like assuming $X + 1 > X$,
and that $(X * 2) / 2 == X$



Out of Bounds Array Accesses

```
int main() {  
    int arr[] = {1,2,3,4,5};  
    return arr[10];  
}
```



Out of Bounds Array Accesses

```
int main() {  
    int arr[] = {1,2,3,4,5};  
    return arr[10];  
}
```

- Possible behaviours: Random value, garbage value, segfault
- Get to skip checking array bounds on every access
- Would need to keep range information with all pointers subject to pointer arithmetic



Case Study 1

```
int main () {  
    struct Struct1* foo = get_struct_or_null();  
    struct Struct2* data = foo->data;  
    if (!foo)  
        return;  
    // ... do stuff using data ...  
}
```

An optimizing compiler would perform the following case analysis:

Case 1: `foo == NULL`

`foo->data` has undefined behavior → Compiler has no particular obligations

Case 2: `foo != NULL`

Null pointer check won't fail → Null pointer check is dead code and may be deleted

Conclusion: remove the if statement

Solution: use compiler flags to turn off optimization



Case Study 2

```
int stupid (int a) {  
  
    return (a+1) > a;  
  
}
```

Case 1: $a \neq \text{INT_MAX}$

Behavior of $+$ is defined \rightarrow Compiler is obligated to return 1

Case 2: $a == \text{INT_MAX}$

Behavior of $+$ is undefined \rightarrow Compiler has no particular obligations

Machine code emitted by an unoptimized compiler:

```
stupid:  
    leal 1(%rdi), %eax  
    cmpl %edi, %eax  
    setg %al  
    movzbl %al, %eax  
    ret
```

Machine code emitted by an optimized compiler?

```
stupid:  
    movl $1, %eax  
    ret
```

Tools for Managing Undefined Behaviour





UBSan (The Undefined Behavior Sanitizer)

- Modify the program at compile-time to catch various kinds of undefined behavior (especially those are non-memory-related) during program execution
- Built into gcc and clang
- Add runtime overhead (typical overhead of 20%)



UBSan (The Undefined Behavior Sanitizer)

- Compiler inserts code that perform certain kinds of checks before operations that may cause undefined behaviors

Before

```
int32_t n = 42;  
n += 17;
```

After

```
int32_t n = 42;  
if (SignedAdditionWillOverflow(n, 17)) {  
    DiagnoseUndefinedBehavior();  
}  
n += 17;
```

UBSan For Dereferencing NULL Pointer

```
> cat null_pointer.c
....
struct foo { int a; int b; };

int main(void) {
    struct foo *x = NULL;
    int m = x->a;
    return 0;
}

> clang -fsanitize=undefined null_pointer.c -o null_pointer

> ./null_pointer
null_pointer.c:10:14: runtime error: member access within null pointer of type 'struct foo'
null_pointer.c:10:14: runtime error: load of null pointer of type 'int'
UndefinedBehaviorSanitizer:DEADLYSIGNAL
==59329==ERROR: UndefinedBehaviorSanitizer: SEGV on unknown address 0x000000000000 (pc
0x0000004229b4 bp 0x7ffc89522840 sp 0x7ffc89522820 T59329)
==59329==The signal is caused by a READ memory access.
....
```

UBSan For Division By Zero

```
> cat division_by_zero.c
....
int main(void) {
    int n = 1;
    int m = n / (n - n);
    return 0;
}

> clang -fsanitize=undefined division_by_zero.c -o division_by_zero

> ./division_by_zero
division_by_zero.c:5:12: runtime error: division by zero
UndefinedBehaviorSanitizer:DEADLYSIGNAL
==4784==ERROR: UndefinedBehaviorSanitizer: FPE on unknown address 0x0000004229db (pc
0x0000004229db bp 0x7ffcdec73880 sp 0x7ffcdec73860 T4784)
    #0 0x4229da in main (/home/z/zpwu17/cs509/division_by_zero+0x4229da)
    #1 0x7fc8e7ee9f89 in __libc_start_main (/lib64/libc.so.6+0x20f89)
    #2 0x402db9 in _start
....
```



Valgrind

- Check memory-related errors
- Translate the input program into an equivalent version that has additional checking. For the memcheck tool, this means it literally looks at the x86 code in the executable, and detects what instructions represent memory accesses.
- Intercept and record the client's malloc, free, calls, etc
- Do not require recompilation
- Add significant runtime overhead



Valgrind For Uninitialized Values

```
> cat uninitialized_values.c
```

```
....
```

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

```
> valgrind ./uninitialized_values
```

```
....
```

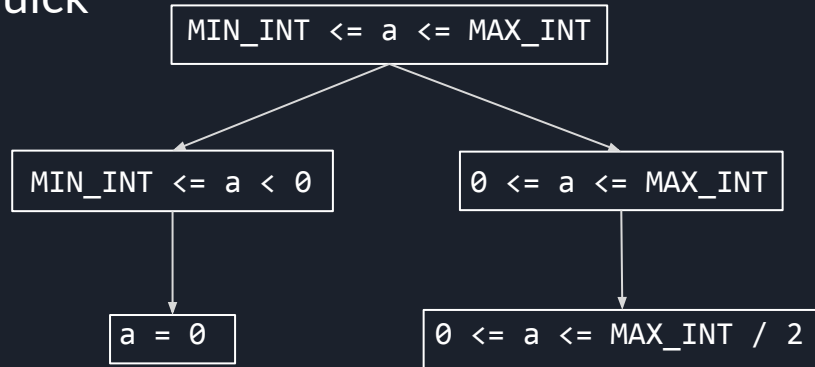
```
==14309== Conditional jump or move depends on uninitialised value(s)  
==14309==    at 0x4EB775A: _IO_file_overflow@@GLIBC_2.2.5 (in /lib64/libc-2.26.so)  
==14309==    by 0x4E8C31D: vfprintf (in /lib64/libc-2.26.so)  
==14309==    by 0x4E925F5: printf (in /lib64/libc-2.26.so)  
==14309==    by 0x400549: main (in /home/z/zpwu17/cs509/uninitialized_values)
```

```
....
```

KLEE Symbolic Analysis Tool

Symbolic execution super quick

```
int foo(int a) {  
    if (a < 0) {  
        a = 0;  
    } else {  
        a = a / 2;  
    }  
    return a;  
}
```





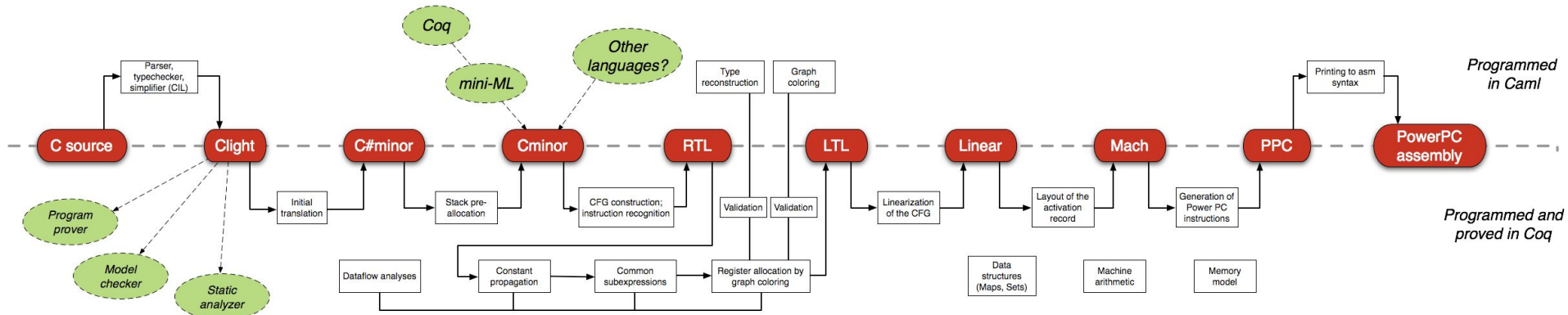
KLEE Symbolic Analysis Tool

- Static analysis to determine input possibilities for test cases
 - Runs on LLVM bitcode
- Can model environment (files, network activity)
- Create unit tests for errors/crashes
- Useful for locating critical failures*

* In the form of crashes, *not* bad results

The CompCert Compiler

- Formally verified compiler, geared towards critical embedded systems
- Source language is Clight, a (very close) subset of C99





The CompCert Compiler

- This provides a couple methods of stability improvements:
 - Clight has definition for some previously undefined behaviour (e.g. data type sizes, dereferencing NULL pointers)
 - Compiler won't incorrectly reorder statements
- This comes at the downside of, unsurprisingly, speed
 - Similar to gcc -O1

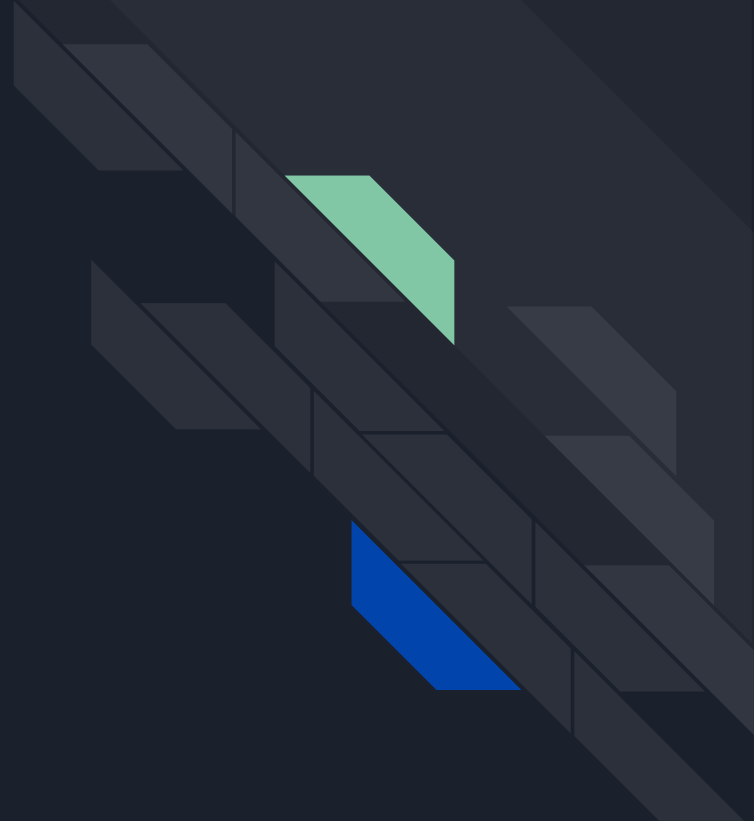


In Summary

- There are pros and cons of undefined behaviour
 - It can cause programmers a lot of issues
 - But it helps make things fast, and easier on compiler writers
- Finding sources of undefined behaviour can be challenging
- Various tools exist to help with undefined behaviour
 - But no silver bullet

Thank You

Questions?



References

- Hathhorn, Ellison, Rosu (2015) Defining the undefinedness of C
- Wang et al. (2012) Undefined behavior: what happened to my code?
- Krebbers (2015) The C standard formalized in Coq
- Leroy (2008) Formal verification of a realistic compiler
- Lattner, What Every C Programmer Should Know About Undefined Behavior,
<http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html>
- Regehr, A Guide to Undefined Behavior in C and C++, Part 1,
<https://blog.regehr.org/archives/213>
- UBSan <https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html>
- Valgrind <http://valgrind.org/>
- KLEE <https://klee.github.io/>
- CompCert <http://compcert.inria.fr/>