Software Reliability

Lecture 12

Compilers and Undefined Behaviour

Alastair Donaldson
www.doc.ic.ac.uk/~afd

Agenda

Examples of undefined behaviours, and surprising interaction with compiler optimisations

Undefined behaviour from the compiler’s perspective

Impact of compilers and undefined behaviour on program analysis

Sources

- John Regehr: “A Guide to Undefined Behaviour in C and C++, Part 1”, http://blog.regehr.org/archives/213
- Xi Wang Haogang Chen Alvin Cheung Zhihao Jia, Nickolai Zeldovich M. Frans Kaashoek: “Undefined Behavior: What Happened to My Code?”, ApSys 2012
- Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, Armando Solar-Lezama: “Towards Optimization-Safe Systems: Analyzing the Impact of Undefined Behavior”, SOSP 2013
- The gcc and clang compilers

Example: saturating add

Let’s try to implement saturating addition for signed integers in C

`x + y` is clamped to an extreme value if it falls outside the signed integer range

```
int saturating_add(int x, int y) {
  if(x > 0 && y > 0 && x + y < 0)
    return INT_MAX;
  if(x < 0 && y < 0 && x + y > 0)
    return INT_MIN;
  return x + y;
}
```

Saturating add in action

Compiled with gcc 4.9.3, -O0, we get:

```
saturating_add(1, 2) == 3
saturating_add(-5, 2) == -3
saturating_add(1000000000, 1000000000) == 2000000000
saturating_add(2000000000, 2000000000) == 2147483647
saturating_add(-2000000000, -2000000000) == -2147483648
```

Intended saturating behaviour (32-bit)

Compiled with gcc 4.9.3, -O2, we get:

```
saturating_add(1, 2) == 3
saturating_add(-5, 2) == -3
saturating_add(1000000000, 1000000000) == 2000000000
saturating_add(2000000000, 2000000000) == -294967296
saturating_add(-2000000000, -2000000000) == 294967296
```

This looks like wrap-around behaviour!

Saturating addition: which results are correct?

Without optimisations, we got:

```
saturating_add(2000000000, 2000000000) == 2147483647
```

With optimisations, we got:

```
saturating_add(2000000000, 2000000000) == -294967296
```

Which is right?

Both: the behaviour of a C program is **undefined** for inputs than can cause signed arithmetic overflow

Technically, it would also be fine for the program to print `"Gotcha!"`, or exhibit **any other behaviour**

Exercise: implement saturating add without invoking undefined behaviour

Let’s implement a guarded abs

```
void print_abs(int x) {
  if(abs(x) < 0) {
    printf("Could not compute abs of %d\n", x);
    return;
  }
  printf("abs of %d is %d\n", x, abs(x));
}
```

Compiled with gcc 4.9.3, -O0, we get:

```
print_abs(100)         -> abs of 100 is 100
print_abs(-100)        -> abs of -100 is 100
print_abs(-2147483648) -> Could not compute abs of -2147483648
```

Compiled with gcc 4.9.3, -O2, we get:

```
print_abs(100)         -> abs of 100 is 100
print_abs(-100)        -> abs of -100 is 100
print_abs(-2147483648) -> abs of -2147483648 is -2147483648
```

Guarded abs: which results are correct?

Again, **both**: attempting to apply **abs** to -2^{n}-1, for *n*-bit signed integers, invokes **undefined behaviour**

In the presence of undefined behaviour, **any result is acceptable**

The ramifications of undefined behaviour

For signed integers **`x`** and **`y`**, **`x + y`** invokes undefined behaviour if the result is outside the signed integer range

Common, wrong, belief: undefined behaviour is local, e.g.

```
z = x + y;
```

has the semantics

```
z = (sumIsInSignedRange(x, y) ? x + y : *);
```

The situation is more drastic

Reality: if a C program P invokes undefined behaviour when executed on input I, there are **no guarantees** about what P will do when executed on input I

C FAQ definition of undefined behaviour

“Anything at all can happen; the Standard imposes no requirements. The program may fail to compile, or it may execute incorrectly (either crashing or silently generating incorrect results), or it may fortuitously do exactly what the programmer intended.”

http://c-faq.com/ansi/undef.html

Common misconception

From John Regehr's blog:

It is very common for people to say something like:

> *"The x86 ADD instruction is used to implement C's signed add operation, and it has two's complement behavior when the result overflows. I'm developing for an x86 platform, so I should be able to expect two's complement semantics when 32-bit signed integers overflow."*

THIS IS WRONG. You are saying something like this:

> *"Somebody once told me that in basketball you can't hold the ball and run. I got a basketball and tried it and it worked just fine. He obviously didn't understand basketball."*

(Explanation due to Roger Miller via Steve Summit: http://www.eskimo.com/~scs/readings/undef.950311.html)

Why is this a misconception?

"The x86 ADD instruction is used to implement C's signed add operation, and it has two's complement behavior when the result overflows. I'm developing for an x86 platform, so I should be able to expect two's complement semantics when 32-bit signed integers overflow."

The programming language has **rules**

The compiler assumes you have obeyed the rules when optimising your code

If you break the rules: compiler optimisations may cause your code to behave unexpectedly (from your perspective) *regardless of what the hardware does*

Is the compiler misbehaving?

No.

An optimising compiler's job:

1. Generate code that respects the language specification
2. Generate efficient code

A compiler can assume your program does **not** exhibit UBs

If your program **does** exhibit UBs then **any behaviour is acceptable**, so whatever the compiler produces satisfies (1) above

The compiler can thus assume no UBs and focus on (2)

The compiler’s “thought process”

“I will assume this program does not exhibit undefined behaviours, because if it does then it matters not what code I emit.”

```
int saturating_add(int x, int y) {
  if(x > 0 && y > 0 && x + y < 0)
    return INT_MAX;
  if(x < 0 && y < 0 && x + y > 0)
    return INT_MIN;
  return x + y;
}
```

“I know **`x + y`** does not overflow: this would be an UB.

So if **`x`** and **`y`** are positive, **`x + y`** must be positive.

The condition is equivalent to *false*.

Excellent!!”

“By similar reasoning, this condition is equivalent to *false.*”

The compiler’s “thought process”

“I can simplify the program to:”

```
int saturating_add(int x, int y) {
  if(false)
    return INT_MAX;
  if(false)
    return INT_MIN;
  return x + y;
}
```

“Or better still, to:”

```
int saturating_add(int x, int y) {
  return x + y;
}
```

Full marks, compiler

Now what happens at runtime?

This is our optimised program:

```
int saturating_add(int x, int y) {
  return x + y;
}
```

For x86, the compiler generates an add instruction that **does** wrap around

Ironically, this is what we anticipated addition would do in our original **`saturating_add`**

Compiler optimisations + wrapping add explains why:

`saturating_add(2000000000, 2000000000)`

gives **`-294967296`**

Compiler's "thought process" again

```
void print_abs(int x) {
  if(abs(x) < 0) {
    printf("Could not compute abs of %d\n", x);
    return;
  }
  printf("abs of %d is %d\n", x, abs(x));
}
```

"**`abs(x)`** is undefined for -2^{32}-1, and otherwise it is non-negative.

I am assuming there are no UBs, so I can assume **`abs(x) < 0`** is *false*, and optimise the program to:"

```
void print_abs(int x) {
  printf("abs of %d is %d\n", x, abs(x));
}
```

At runtime: x86 instructions generated for **`abs(-`2^{32}`-1)`** happen to give **`-`2^{31}`-1`**

A security-critical example

```
unsigned int
tun_chr_poll(struct file *file, poll_table * wait)
{
  struct tun_file *tfile = file->private_data;
  struct tun_struct *tun = __tun_get(tfile);
  struct sock *sk = tun->sk;
  if (!tun)
    return POLLERR;
  ...
}
```

Programmer's reasoning: if **`tun`** is null, either:

– **`tun->sk`** will cause an abort

or (if address 0 is mapped)

– **`!tun`** will hold, and **`POLLERR`** will be returned

Compiler's "thought process"

```
unsigned int
tun_chr_poll(struct file *file, poll_table * wait)
{
  struct tun_file *tfile = file->private_data;
  struct tun_struct *tun = __tun_get(tfile);
  struct sock *sk = tun->sk;
  if (!tun)
    return POLLERR;
  ...
}
```

"Thus I can optimise away this block of code!"

"The programmer is telling me that **`tun`** is not null (because otherwise **`tun->sk`** invokes undefined behaviour)"

If **`tun`** is 0 but **`tun->sk`** does not abort, execution continues with a bad **`struct sock`**. This was the basis for a privileged escalation exploit, see: http://lwn.net/Articles/342330/

Why do languages have undefined behaviour?

Reason 1: to cater for diverse hardware

Example: signed overflow

- x86: signed addition wraps on overflow
- MIPS: signed addition traps on overflow
- UNISYS 2200 (legacy): ones complement representation of integers

Example: shifting

- Left-shifing 1 by 32 bits, in 32-bit register…
- Produces 0 on ARM and PowerPC
- Produces 1 on x86

Why do languages have undefined behaviour?

Example: dereferencing null

- On x86, dereferencing 0 usually causes runtime exception
- …but one can also memory-map 0 to a valid page
- On ARM, address 0 holds exception handlers

Standardising behaviour would be **expensive** for some or all platforms, e.g.:

- Specifying wrap-around behaviour for signed addition would be free on x86, but would require checking code on MIPS
- Defining over-shifting to yield 0 would require compiling **`x << y`** to **`(y > 31 ? 0 : x << y)`**

Why do languages have undefined behaviour?

Reason 2: to allow powerful optimisations

Consider:

```
bool find(int *A, int x, int n) {
  for(int i = 0; i <= n; i++)
    if(A[i] == x)
      return true;
  return false;
}
```

On 64-bit architecture, pointers are 64-bit, but **`int`** is 32-bit

Means that **`A[i]`** requires **`i`** to be extended to 64-bit

Optimisation: change type of **`i`** to **`long`**, which is 64-bit

Only works if i++ does not overflow

Making signed overflow **undefined** allows this optimisation

More undefined behaviours in C

There are lots. Examples include:

- Reading from uninitialised variables
- Pointing past the end of an array (even if you do not dereference the pointer)
- Aliasing between pointers with different types
- Calling `memcpy` with buffers that overlap

Key point to remember:

The compiler optimises your code under the assumption that **no UBs are invoked**

Impact of undefined behaviour on program analysers

- Many program analysis tools to re-use compiler front-ends: saves **enormous** front-end development effort
- Two examples:
  - **GPUVerify** uses Clang to translate OpenCL kernels into LLVM byte-code, GPUVerify then works on the LLVM byte-code
  - **SMACK** translates LLVM byte-code into the Boogie verification language. A successful C-based analyser uses Clang to convert C to LLVM, and then applies SMACK
- Problem: the compiler front-end can generate **arbitrary** code for inputs that exhibit UBs

Example: GPUVerify and UBs

- GPUVerify says that this kernel is *race-free*

```
kernel void foo(global int * A) {
  if(2/0 == 2) {
    A[0] = get_global_id(0);
  }
}
```

- **Reason:** the Clang front-end regards the write to `A` as unreachable, since it would require a UB in order to be reached
- **Lesson:** relying on compiler front-ends that exploit UBs can render a would-be sound tool **unsound**

(In this case Clang does at least warn about the UB)

PhD opportunities

- If you have enjoyed the course and are considering doing a PhD then please talk to me and/or Cristian about possible PhD opportunities!
- Talk to us after a lecture or drop an email

A concluding detour from Ally

What is better:

An over-approximating tool that **quickly reports all possibly buggy lines of code**, but has a **high false positive rate**

An under-approximating tool that **quickly reports successful verification for all correct programs** but has a **high false negative rate**

An exact tool that produces **no false positives or false negatives**, but has a **high rate of non-termination**

?

Amazing analyser #1

Under-approximating analyser (in Python):

```
print “The input program is correct!”
```

Very efficient!

Never reports a false positive!!

Works for any programming language!!!

Drawback: not so useful for finding bugs

Managers like this tool

Amazing analyser #2

Over-approximating analyser (in Python):

```
if len(input_file) == 0:
  print “The input program is correct!”
for i in range(1, len(input_file)+1):
  print “Possible error at line ” + str(i)
```

Very efficient!

Never reports a false negative!!

Works for any programming language!!!

Drawback: has only ever managed to verify one program

Amazing analyser #3

Exact analyser that may not terminate (in Python):

```
if len(input_file) == 0:
  print “The input program is correct!”
while true:
  pass
```

Never reports a false positive or negative!

Works for any programming language!!

Has only ever managed to verify one program **so far**, but **experiments on other programs are still running**

Back to our trick question

It is easy to write a **useless** analyser that satisfies any of the stated requirements

In general: no way to say whether a sound technique that produces false positives is better than a bug-finding method which reports false negatives

Key is to design a careful combination of analyses that works effectively in a particular domain