Dude, where's my code?

Towards Optimization-Safe Systems

Analyzing the Impact of Undefined Behavior

Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama
MIT CSAIL

Belief: compiler == faithful translator

C

```c
int main()
{
    ...
    return 0;
}
```

X86

```
...
movq %rsp, %rbp
xorl %eax, %eax
popq %rbp
ret
```

Not true if your code invokes undefined behavior

- Security implications

Example: compiler discards sanity check

```
char *buf        = ...;
char *buf_end    = ...;
unsigned int off = /* read from untrusted input */;
if (buf + off >= buf_end)
  return;          /* validate off: buf+off too large*/
if (buf + off < buf)
  return;          /* validate off: overflow, buf+off wrapped around */
/* access buf[0..off-1] */
```

- C spec: pointer overflow is undefined behavior
  - gcc: `buf + off` cannot overflow, different from hardware!
  - gcc: `if (buf + off < buf)` ⇒ `if (false)`
- Attack: craft a large `off` to trigger buffer overflow

Undefined behavior allows such optimizations

Undefined behavior: the spec "imposes no requirements"

- Original goal: emit efficient code
- Compilers assume a program never invokes undefined behavior
- Example: no bounds checks emitted; assume no buffer overflow

```
*p = 42;          /* store 42 to p */

mov $42, (%rdi)   /* no bounds checks */
```

Examples of undefined behavior in C

Meaningless checks from real code: pointer p; signed integer x

Pointer overflow:	`if (p + 100 < p)`
Signed integer overflow:	`if (x + 100 < x)`
Oversized shift:	`if (!(1 << x))`
Null pointer dereference:	`*p; if (p)`
Absolute value overflow:	`if (abs(x) < 0)`

Problem: unstable code confuses programmers

Unstable code: compilers discard code due to undefined behavior

C

```c
...
if (p + 100 < p)
  return;
```

GCC

X86

```
...
nop
nop
```

- Security checks discarded
- Weakness amplified
- Unpredictable system behavior

Contributions

- A case study of unstable code in real world
- An algorithm for identifying unstable code
- A static checker STACK
  - 160 previously unknown bugs confirmed and fixed
  - Users: Intel, several open-source projects, ...

Example: broken check in Postgres

Implement 64-bit signed division `x/y` in SQL

```
if (y == -1 && x < 0 && (x / y < 0))   /* -2^63/-1 < 0? */
  error();
```

- Some compilers optimize away the check
- x86-64's `idivq` traps on overflow: DoS attack

```sql
SELECT ((-9223372036854775808)::int8) / (-1);
```

SQL

Example: fix check in Postgres

Our proposal:

```
if (y == -1 && x == INT64_MIN) /* INT64_MIN is -2^63*/
```

Developer's fix:

```
if (y == -1 && ((-x < 0) == (x < 0)))
```

- Still unstable code: time bomb for future compilers
  - "it's an overflow check so it should check for overflow"
  - "we don't want the constant `INT64_MIN`; it's less portable"

“This will create MAJOR SECURITY ISSUES in ALL MANNER OF CODE. I don’t care if your language lawyers tell you gcc is right. . . . FIX THIS! NOW!”

a gcc user
bug #30475 - assert(int+100 > int) optimized away

“I am sorry that you wrote broken code to begin with . . . GCC is not going to change.”

a gcc developer
bug #30475 - assert(int+100 > int) optimized away

Test existing compilers

12 C/C++ compilers

- gcc
- aCC (HP)
- icc (Intel)
- open64 (AMD)
- suncc (Oracle)
- ti (TI's TMS320C6000)
- clang
- armcc (ARM)
- msvc (Microsoft)
- pathcc (PathScale)
- xlc (IBM)
- windriver (Wind River's Diab)

Examples of unstable code

Meaningless checks from real code: pointer p; signed integer x

Pointer overflow:	`if (p + 100 < p)`	⇒ `if (false)`
Signed integer overflow:	`if (x + 100 < x)`	⇒ `if (false)`
Oversized shift:	`if (!(1 << x))`	⇒ `if (false)`
Null pointer dereference:	`*p; if (p)`	⇒ `if (false)`
Absolute value overflow:	`if (abs(x) < 0)`	⇒ `if (false)`

Compilers often discard unstable code

	`if(p+100<p)`	`if(x+100<x)`	`if(!(1<<x))`	`*p; if(!p)`	`if(abs(x)<0)`
gcc-4.8.1	O2	O2		O2	O2
clang-3.3	O1	O1	O1		
aCC-6.25					O3
armcc-5.02		O2			
icc-14.0.0		O1		O2	
msvc-14.0.0				O1	
open64-14.0.0	O1	O2			O2
pathcc-1.0.0	O1	O2			O2
suncc-5.12				O3	
ti-7.4.2	O0	O0			
windriver-5.9.2		O0			
xlc-12.1	O3				

Compilers become more aggressive over time

	`if(p+100<p)`	`if(x+100<x)`	`if(!(1<<x))`	`*p; if(!p)`	`if(abs(x)<0)`
(1992) gcc-1.42					
(2001) gcc-2.95.3		O1			
(2006) gcc-3.4.6		O1		O2	
(2007) gcc-4.2.1	O0	O2			O2
(2013) gcc-4.8.1	O2	O2		O2	O2
(2009) clang-1.0	O1				
(2010) clang-2.8	O1	O1			
(2013) clang-3.3	O1	O1	O1		

Observation

- Compilers silently remove unstable code
- Different compilers behave in different ways
  - Change/upgrade compiler $\Rightarrow$ broken system
- Need a systematic approach

Our approach: precisely flag unstable code

C/C++ source → LLVM IR → STACK → warnings

```
% ./configure
% stack-build make         # intercept cc & generate LLVM IR
% poptck                   # run STACK in parallel
```

STACK provides informative warnings

```
1. res = x / y;
2. if (y == -1 && x < 0 && res < 0)
3.   return;
```

The check at line 2 is simplified into false, due to division at line 1

```
model: |                          # possible optimization
  %cmp3 = icmp slt i64 %res, 0
  -->  false
stack:                            # location of unstable code
  - div.c:2
core:                             # why optimized away
  - div.c:1
    - signed division overflow
```

Design overview of STACK

- What's the difference, compilers vs most programmers?
  - Assumption Δ: programs don't invoke undefined behavior
- What can compilers do *only* with assumption Δ?
  - Optimize away unstable code
- STACK: mimic a compiler that selectively enables Δ
  - Phase I: optimize w/o Δ
  - Phase II: optimize w/ Δ
  - Unstable code: difference between the two phases

Example of identifying unstable code

```
res = x / y;
if (y == -1 && x < 0 && res < 0)
  return;
```

- Assumption Δ:
  - No division by zero: $y \neq 0$
  - No division overflow: $y \neq -1$ OR $x \neq$ INT_MIN
- STACK *can* optimize “`res < 0`” to “`false`” only with Δ
  - Phase I: is “`res < 0`” equivalent to “`false`” in general? No.
  - Phase II: is “`res < 0`” equivalent to “`false`” with Δ? Yes!
- Report “`res < 0`” as unstable code

Compute assumption Δ

One must *not* trigger undefined behavior at any code fragment

- `Reach(e)`: when to reach and execute code fragment e
- `Undef(e)`: when to trigger undefined behavior at e

```
Δ = ∀e: Reach(e) → ¬Undef(e)
```

Example: compute assumption Δ

One must not trigger undefined behavior at any code fragment

```
Δ = ∀e: Reach(e) → ¬Undef(e)
```

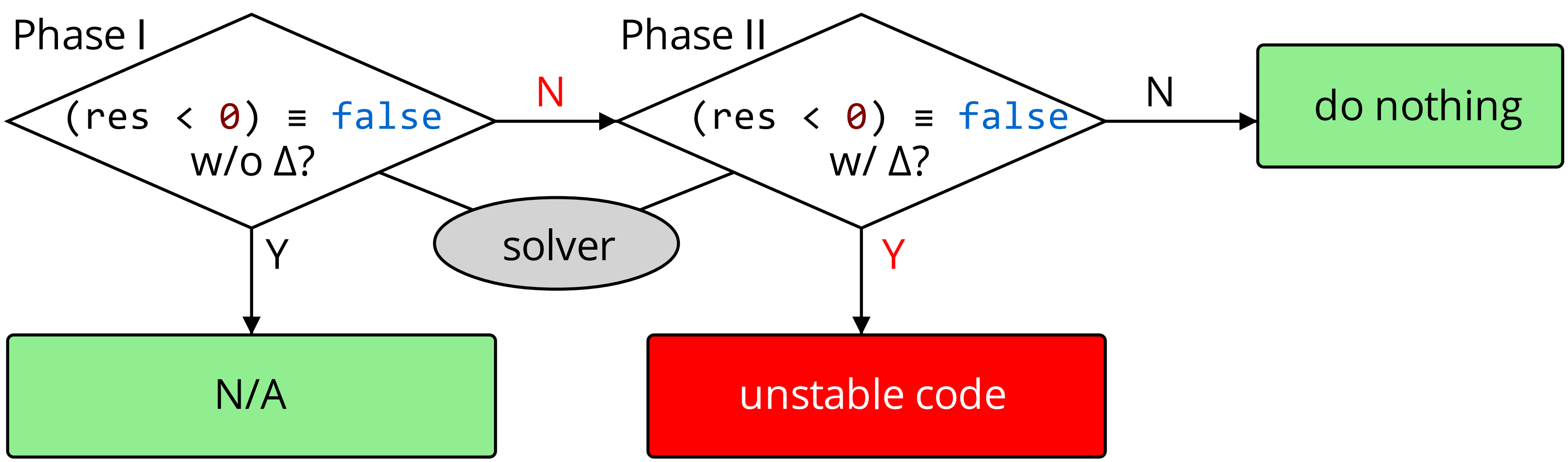
```
1. res = x / y;
2. if (y == -1 && x < 0 && res < 0)
3.   return;
```

```
Δ = true → ¬((y == 0) ∨ (x == -1 ∧ y == INT_MIN)) # line 1
  ∧ true → ¬false                                  # line 2
  ∧ ((y == -1) ∧ (x < 0) ∧ (x/y < 0)) → ¬false     # line 3

Δ = ¬((y == 0) ∨ (x == -1 ∧ y == INT_MIN))
```

Find unstable code by selectively enabling Δ

```
1. res = x / y;
2. if (y == -1 && x < 0 && res < 0)
3.   return;
```



Summary of STACK

- Compute assumption Δ: no undefined behavior
- Two-phase framework: w/o and w/ Δ
  - Report unstable code from difference
- Limitations
  - Missing unstable code: Phase II not powerful enough
  - False warnings: Phase I not powerful enough

Implementation of STACK

- LLVM
- Boolector solver
- ~4,000 lines of C++ code
- Per-function for better scalability
  - Could miss bugs

Evaluation

- Is STACK useful for finding unstable code?
- How precise are STACK's warnings?
- How prevalent is unstable code?
- How much time to analyze a large code base?

STACK finds new bugs

- Applied STACK to many popular systems
- Inspected warnings and submitted patches to developers
  - Binutils, Bionic, Dune, e2fsprogs, FFmpeg+Libav, file, FreeType, GMP, GRUB, HiStar, Kerberos, libX11, libarchive, libgcrypt, Linux kernel, Mosh, Mozilla, OpenAFS, OpenSSH, OpenSSL, PHP, plan9port, Postgres, Python, QEMU, Ruby+Rubinius, Sane, uClibc, VLC, Wireshark, Xen, Xpdf
- Developers accepted most of our patches
  - 160 new bugs

STACK warnings are precise

- Kerberos: STACK produced 11 warnings
  - Developers accepted every patch
  - No warnings for fixed code
  - Low false warning rate: 0/11
- Postgres: STACK produced 68 warnings
  - 9 patches accepted: server crash
  - 29 patches in discussion: developers blamed compilers
  - 26 time bombs: can be optimized away by future compilers
  - 4 false warnings: benign redundant code
  - Low false warning rate: 4/68

Unstable code is prevalent

- Applied STACK to all Debian Wheezy packages
  - 8,575 C/C++ packages
  - ~150 days of CPU time to build and analyze
- STACK warns in ~40% of C/C++ packages

STACK scales to large code bases

Intel Core i7-980 3.3 GHz, 6 cores

	build time	analysis time	# files
Kerberos	1 min	2 min	705
Postgres	1 min	11 min	770
Linux kernel	33 min	62 min	14,136

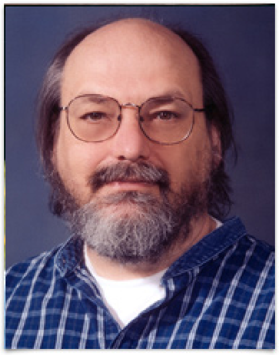
How to avoid unstable code

- Programmers
  - Fix bugs
  - Workaround: disable certain optimizations
- Compilers & checkers
  - Many bug-finding tools fail to model C spec correctly
  - Use our ideas to generate better warnings
- Language designers: revise the spec
  - Eliminate undefined behavior? Perf impact?

Other application

Reflections on trusting trust [Thompson84]

- Hide backdoors
  - Submit a new feature with unstable code
  - Could easily slip through code review



Summary

- Compilers optimize away unstable code
  - Subtle bugs
  - Significant security implications
- Compiler writers: use our techniques to generate better warnings
- Language designers: trade-off between performance & security
- Programmers: check your C/C++ code using STACK

http://css.csail.mit.edu/stack/

Q: CPU emulator

16-bit multiplication, from a well-known company

```
uint64_t mul(uint16_t a, uint16_t b)
{
    uint32_t c = a * b;
    return c;
}
```

What's the result of `mul(0xffff, 0xffff)`?

a) `1`

b) `0xfffe0001`

c) `0xfffffffffffe0001`