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

Not true if your code invokes undefined behavior

- Security implications
Example: compiler discards sanity check

```c
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

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

```assembly
mov $42, (%rdi)  /* no bounds checks */
```
Examples of undefined behavior in C

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

- Pointer overflow: \( \text{if} \ (p + 100 < p) \)
- Signed integer overflow: \( \text{if} \ (x + 100 < x) \)
- Oversized shift: \( \text{if} \ (! (1 << x)) \)
- Null pointer dereference: \( \star p; \ \text{if} \ (p) \)
- Absolute value overflow: \( \text{if} \ (\text{abs}(x) < 0) \)
Problem: unstable code confuses programmers

*Unstable code*: compilers discard code due to undefined behavior

- 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

```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);
```
Example: fix check in Postgres

Our proposal:

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

Developer’s fix:

```c
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: \[ \text{if } (p + 100 < p) \Rightarrow \text{if } (\text{false}) \]
Signed integer overflow: \[ \text{if } (x + 100 < x) \Rightarrow \text{if } (\text{false}) \]
Oversized shift: \[ \text{if } (\neg (1 << x)) \Rightarrow \text{if } (\text{false}) \]
Null pointer dereference: \[ \ast p; \text{if } (p) \Rightarrow \text{if } (\text{false}) \]
Absolute value overflow: \[ \text{if } (\text{abs}(x) < 0) \Rightarrow \text{if } (\text{false}) \]
Compilers often discard unstable code

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Optimization Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc-4.8.1</td>
<td>02</td>
</tr>
<tr>
<td>clang-3.3</td>
<td>01</td>
</tr>
<tr>
<td>aCC-6.25</td>
<td>01</td>
</tr>
<tr>
<td>armcc-5.02</td>
<td>02</td>
</tr>
<tr>
<td>icc-14.0.0</td>
<td>01</td>
</tr>
<tr>
<td>msvc-14.0.0</td>
<td>01</td>
</tr>
<tr>
<td>open64-14.0.0</td>
<td>01</td>
</tr>
<tr>
<td>pathcc-1.0.0</td>
<td>02</td>
</tr>
<tr>
<td>suncc-5.12</td>
<td>03</td>
</tr>
<tr>
<td>ti-7.4.2</td>
<td>00</td>
</tr>
<tr>
<td>windriver-5.9.2</td>
<td>00</td>
</tr>
<tr>
<td>xlc-12.1</td>
<td>03</td>
</tr>
</tbody>
</table>

where:  
if(p+100<p)  
if(x+100<x)  
if(!(1<<x))  
*p; if(!p)  
if(abs(x)<0)
Compilers become more aggressive over time

<table>
<thead>
<tr>
<th>Year</th>
<th>Compiler</th>
<th>Optimization Level</th>
<th>Year</th>
<th>Compiler</th>
<th>Optimization Level</th>
<th>Year</th>
<th>Compiler</th>
<th>Optimization Level</th>
<th>Year</th>
<th>Compiler</th>
<th>Optimization Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>1992</td>
<td>gcc-1.42</td>
<td></td>
<td>2001</td>
<td>gcc-2.95.3</td>
<td>O1</td>
<td>2006</td>
<td>gcc-3.4.6</td>
<td>O1</td>
<td>2007</td>
<td>gcc-4.2.1</td>
<td>O0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2009</td>
<td>clang-1.0</td>
<td>O1</td>
<td>2010</td>
<td>clang-2.8</td>
<td>O1</td>
<td>2013</td>
<td>clang-3.3</td>
<td>O1</td>
<td>2013</td>
<td>clang-3.3</td>
<td>O1</td>
</tr>
</tbody>
</table>
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: |
  %cmp3 = icmp slt i64 %res, 0
  --> false
stack:  # possible optimization
  - div.c:2
core:  # location of unstable code
  - div.c:1
    - signed division overflow
  - 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

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

- Assumption Δ:
  - No division by zero: y ≠ 0
  - No division overflow: y ≠ -1 OR x ≠ 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 $\Delta$

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

- $\text{Reach}(e)$: when to reach and execute code fragment $e$
- $\text{Undef}(e)$: when to trigger undefined behavior at $e$

$$\Delta = \forall e: \text{Reach}(e) \rightarrow \neg \text{Undef}(e)$$
Example: compute assumption $\Delta$

One must not trigger undefined behavior at any code fragment

$$\Delta = \forall e: \text{Reach}(e) \rightarrow \neg \text{Undef}(e)$$

1. $\text{res} = x / y$;
2. $\text{if } (y == -1 \land x < 0 \land \text{res} < 0)$
3. $\text{return}$;

$$\Delta = \text{true} \rightarrow \neg((y == 0) \lor (x == -1 \land y == \text{INT}_\text{MIN})) \ # \text{line 1}$$
$$\land \text{true} \rightarrow \neg\text{false} \ # \text{line 2}$$
$$\land ((y == -1) \land (x < 0) \land (x/y < 0)) \rightarrow \neg\text{false} \ # \text{line 3}$$

$$\Delta = \neg((y == 0) \lor (x == -1 \land y == \text{INT}_\text{MIN}))$$
Find unstable code by selectively enabling $\Delta$

1. $\text{res} = x / y;$
2. $\text{if } (y == -1 \&\& x < 0 \&\& \text{res} < 0)$
3. $\text{return;}$

Phase I

- $(\text{res} < 0) \equiv \text{false}$ w/o $\Delta$?
  - $\text{Y}$
    - N/A
  - $\text{N}$
    - solver

Phase II

- $(\text{res} < 0) \equiv \text{false}$ w/ $\Delta$?
  - $\text{Y}$
    - unstable code
  - $\text{N}$
    - do nothing
Summary of STACK

- Compute assumption $\Delta$: no undefined behavior
- Two-phase framework: w/o and w/ $\Delta$
  - 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

<table>
<thead>
<tr>
<th></th>
<th>build time</th>
<th>analysis time</th>
<th># files</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kerberos</td>
<td>1 min</td>
<td>2 min</td>
<td>705</td>
</tr>
<tr>
<td>Postgres</td>
<td>1 min</td>
<td>11 min</td>
<td>770</td>
</tr>
<tr>
<td>Linux kernel</td>
<td>33 min</td>
<td>62 min</td>
<td>14,136</td>
</tr>
</tbody>
</table>
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

```c
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) 0xffffe0001
c) 0xfffffffffffffff0001