# CSE 333

## lec 23: undefined behavior

Xi Wang

Department of Computer Science & Engineering

University of Washington

# administrivia

Thursday: hw4 due

Friday: wrapup / Q&A

next Monday: Q&A (560)

next Wednesday: final exam, open book/notes/laptop

# most time-consuming bugs

memory bugs: dangling pointer, double free, “`&var`” vs “`var`”, leak (“800 valgrind warnings”), null pointers (“3 days”), variable lifetime (“1 week”)

logical bugs: wrong functions (“`getnameinfo`”), wrong variables (“`head`” vs “`header`”), wrong seek offset/count

others: missing semicolons, missing parentheses, `size_t` vs `uint32_t`, performance bugs, “no idea what happened”

# ex19 quotes

*The[re] are so many bugs in CSE333 that drove me crazy.*

*I've tried my best to repress most of the pain this class has caused me…*

# undefined behavior

- can lead to unstable code: intended code altered by compilers due to undefined behavior
  - useful code unexpectedly gone
  - not a compiler bug: legal optimizations
  - not a spec bug: spec allows anything to happen
- joint work with Nickolai Zeldovich, Frans Kaashoek, Armando Solar-Lezama

# Unstable code demo: Intel's CPU emulator

C/C++

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

Question: what's the result of `mul(60000, 60000)`?

- (a) 3,600,000,000
- (b) 18,446,744,073,014,584,320
- (c) something else

# Unstable code has serious security implications

- Unstable code ⇒ buffer overflow (full control)
- Unstable code ⇒ denial of service (crash)
- Unstable code ⇒ non-random random numbers

---

http://lists.apple.com/archives/security-announce/2013/Oct/msg00004.html

```
Libc
Impact: Under unusual circumstances some random numbers may be predictable
Description: ...

CVE-2013-5180
```

---

# State of the art

- Wisdom: turn off optimizations if seeing weird bugs
- Blog posts and write-ups
  - Chris Lattner: What every C programmer should know about undefined behavior
  - John Regehr: A guide to undefined behavior in C and C++
  - Robert Seacord: Dangerous optimizations and the loss of causality

# Challenges

- How prevalent?
- How to think about it?
- How to detect?

# Contributions

- How prevalent: major compilers; 160+ new bugs


iOS

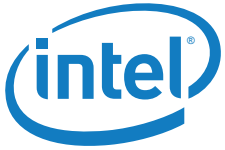

Android


Chrome


Python

- How to think about it: formulated as boolean satisfiability
- How to detect: a practical checker *STACK*; adopted by companies





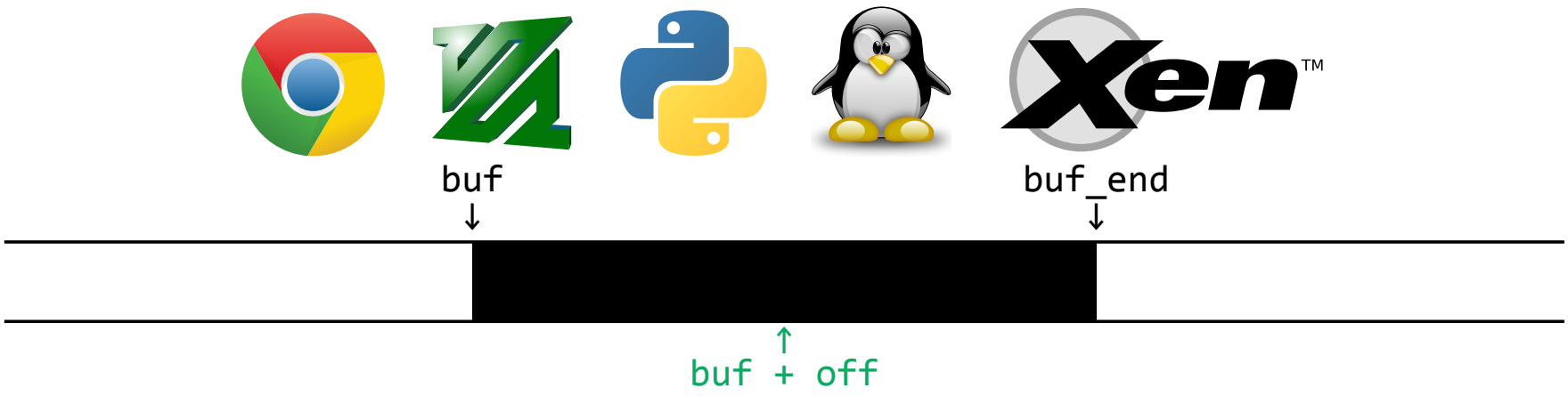

- Influenced C++ committee to form SG12 group

# Part 0: undefined behavior

- What is undefined behavior
- How undefined behavior leads to unstable code

# Attack: unstable code ⇒ buffer overflow

```
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] */
```




- gcc: `buf + off` cannot become smaller (different from hardware!)
  - gcc: `if (buf + off < buf)` ⇒ `if (false)`

# Undefined behavior allows such optimizations

Undefined behavior: the spec "imposes no requirements"

- Original goal: emit efficient code
- Example: division by zero is undefined behavior
  - Spec: program can do anything if that occurs
  - Compiler: no need to emit zero check on divisor

```
x / y  ⇒  div %esi
          /* no zero check on y */
```

- Pointer overflow is undefined behavior, too!
  - Program can do anything if "`buf + off`" overflows
  - gcc: `if (buf + off < buf)` ⇒ `if (false)`

# Examples of undefined behavior in C

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
  - Code may or may not work
  - Depend on compilers (+ hardware/OS)

“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

# Part I: how prevalent?

Test 12 major 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

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)` |

# Major compilers discard unstable code

| | `if(p+100<p)` | `if(x+100<x)` | `if(!(1<<x))` | `*p; if(!p)` | `if(abs(x)<0)` |
|---|---|---|---|---|---|
| gcc-4.9.1 | O2 | O2 | | O2 | O2 |
| clang-3.4 | 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 |
| (2014) gcc-4.9.1 | O2 | O2 | | O2 | O2 |
| (2009) clang-1.0 | O1 | | | | |
| (2010) clang-2.8 | O1 | O1 | | | |
| (2014) clang-3.4 | O1 | O1 | O1 | | |

# No single don't-be-evil optimization option

- Modern compilers are complicated
  - gcc 4.9: -O2 turns on 203/274 mid-end passes
  - Many parts make decisions: interaction and side effects
  - Inlining + constant folding + range + dead code elim
- Consequence: hard to turn "off" one particular optimization

# Unstable code affects a wide range of software

Unstable code found in software written using C/C++

- Higher-level languages: PHP, Python, Ruby
- Applications
  - Web browsing: Chrome
  - Movie decoding: FFmpeg
  - Font rendering: FreeType

# Summary of Part I

Unstable code is an emerging threat

- Programmers have made mistakes (for many years)
- Modern compilers make it worse
- Change/upgrade compiler ⇒ broken system

# Part II: how to think about unstable code

Strawman approach: as dead code

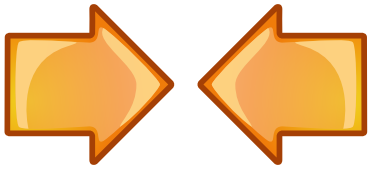
- Ask a compiler to warn whenever it eliminates dead code
- Problems
  - Restricted to one particular compiler
  - Not general: sensitive to optimizations
  - Lots of false warnings: compiler kills dead code all the time

# Our approach: as boolean satisfiability (SAT)

| | compiler independent | general to a bug class | low false warning rate |
|---|---|---|---|
| as dead code | | | |
| New: as SAT | ✓ | ✓ | ✓ |

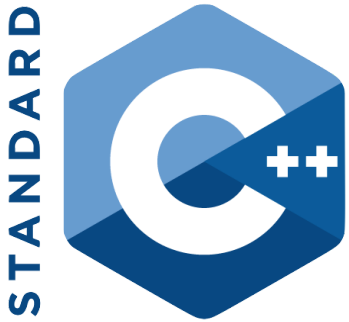
# Cause: disagree on spec (undefined behavior)

programmer: useful code




compiler: dead code

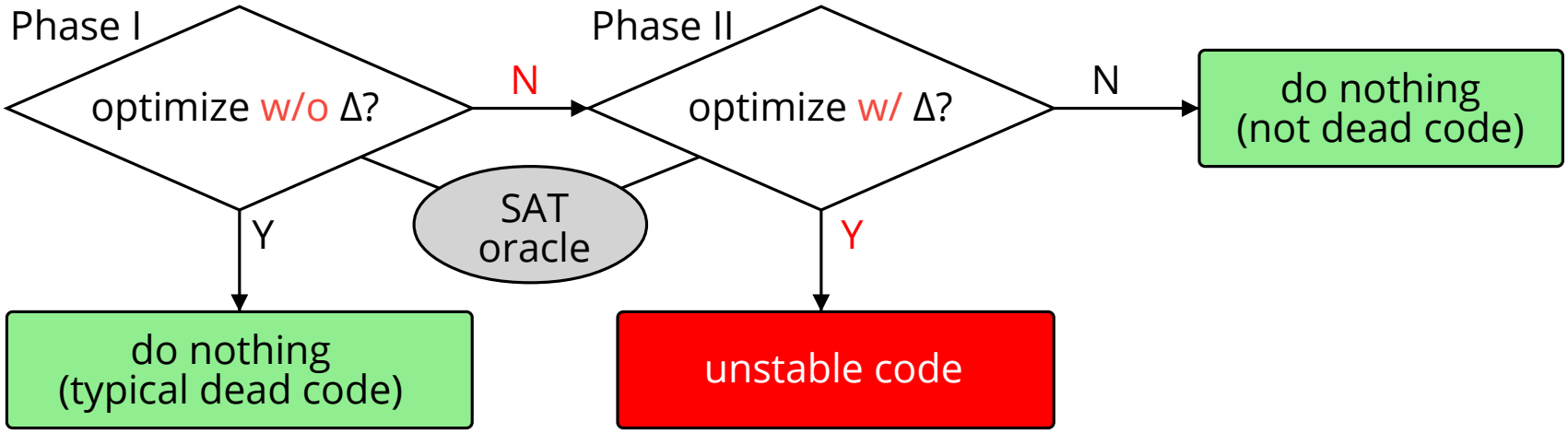
---




spec

# Formulation overview

- Disagreement Δ
  - Compiler: program never invokes undefined behavior
- What can be done only with Δ: kill unstable code
  - Mimic a super aggressive optimizer



# Step 1/2: Finding Δ

```
1.  if (buf + off >= buf_end)
2.    return;
3.  if (buf + off < buf)
4.    return;
```

Δ: what compilers can assume from `buf + off`

- No pointer overflow: `NOT(buf∞ + off∞ > max)`

# Formulate Δ

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

- `Reach(e, in)`: with what input to reach/execute code fragment e
- `Undef(e, in)`: with what input to trigger undefined behavior at e

```
Δ(in) = ∀e: Reach(e, in) → ¬Undef(e, in)
```

# Example: compute Δ

```
1.  if (buf + off >= buf_end)
2.    return;
3.  if (buf + off < buf)
4.    return;
```

Reach Undef

$$\Delta(in) = \wedge_e \text{Reach}(e, in) \rightarrow \neg\text{Undef}(e, in)$$
$$= \neg(buf_\infty + off_\infty > max)$$

# Step 2/2: reason about unstable code with Δ

```
1.  if (buf + off >= buf_end)
2.    return;
3.  if (buf + off < buf)
4.    return;
```

- Is `(buf + off < buf)` equivalent to `false`?
  - SAT oracle: N
- Is `(buf + off < buf)` equivalent to `false` w/ Δ?
  - Δ: $\neg$`(buf`$_\infty$ `+ off`$_\infty$ `> max)`
  - SAT oracle: Y

"`buf + off < buf`" is unstable code

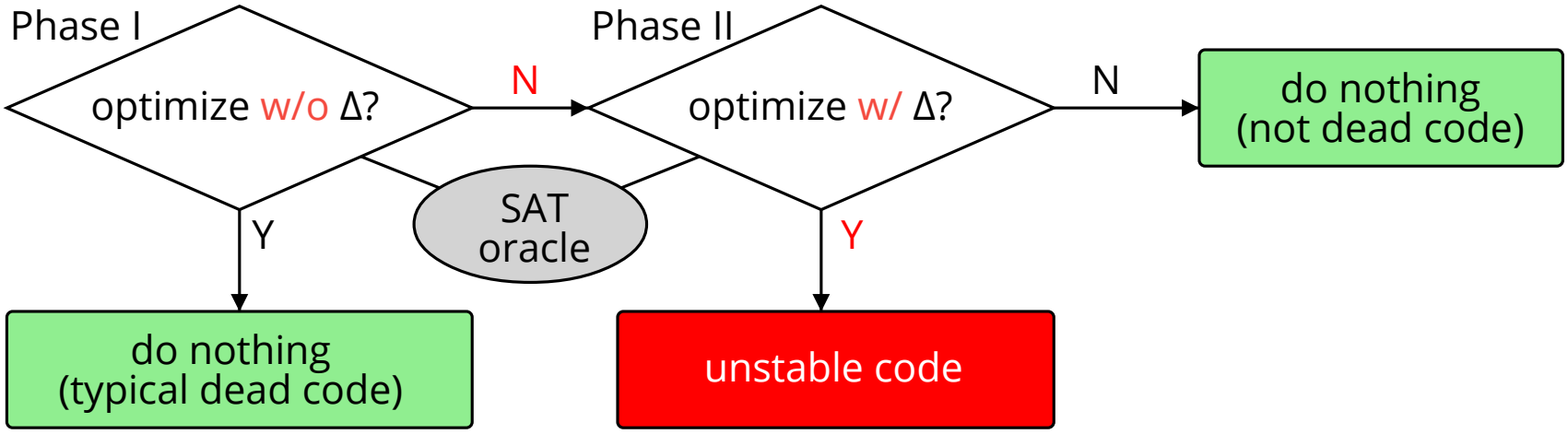
# Find unstable code that can be turned into false

Find every boolean expression `b` that satisfies the following

```
  (∃in: b(in) ≠ false ∧ Reach(b, in))          # 1: not trivially dead code
∧ (∄in: b(in) ≠ false ∧ Reach(b, in) ∧ Δ(in)) # 2: unstable code
```

- Generalize to find unstable code
  - Expressions that can be turned into true only w/ Δ
  - Statements that can become unreachable only w/ Δ

# Understand false & missing errors




- Phase I not powerful enough: false errors (dead code)
- Phase II not powerful enough: missing errors

# Understand unstable CPU emulator

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

Question: what's the result of `mul(60000, 60000)`?

- (a) 3,600,000,000
- (b) 18,446,744,073,014,584,320
- (c) something else

# Summary of Part II

Unstable code as SAT problem

- Formulate disagreement $\Delta$
- Find optimization diff between w/o and w/ $\Delta$
- Compiler-independent, precise, and general

# Part III: how to detect

STACK: unstable code checker

- Practical challenges
- Evaluation of STACK

# Practical challenges

For every code fragment `e` in a program

$$\Delta(in) = \forall e: Reach(e, in) \rightarrow \neg Undef(e, in)$$

Problem: infeasible to compute

- Require to inspect the entire program by definition
- Precision: loops, function pointers, etc.
- Scalability
  - Gigantic boolean predicate: unsolvable
  - Hard to parallelize

# STACK: per-function and approximation

- Analyze each function independently: smaller SAT and parallel
- Careful approximation to maintain high precision
  - One-side error: no illegal optimization
  - Trade-off: could miss bugs

**A Correctness of approximation**

As discussed in §3.2, STACK performs an optimization if the corresponding query $Q$ is unsatisfiable. Using an approximate query $Q'$ yields a correct optimization if $Q'$ is weaker than $Q$ (i.e., $Q \rightarrow Q'$): if $Q'$ is unsatisfiable, which enables the optimization, the original query $Q$ must also be unsatisfiable.

To prove the correctness of approximation, it suffices to show that the approximate elimination query (5) is weaker than the original query (3); the simplification queries (6) and (4) are similar. Formally, given code fragment $e$, it suffices to show the following:

$$R_e(\mathbf{x}) \wedge \Delta(\mathbf{x}) \rightarrow R'_e(\mathbf{x}) \wedge \bigwedge_{d \in \mathrm{dom}(e)} \neg U_d(\mathbf{x}). \qquad (8)$$

...

# Implementation of STACK

- LLVM compiler framework
- Boolector solver
- ~4,000 lines of C++ code

# Easily integrated into development

C/C++ source → STACK → warnings

```
% ./configure
% stack-build make  # intercept compiler invocation & dump data for analysis
% poptck            # run checker in parallel
```

# STACK provides informative warnings

```
1.  if (buf + off >= buf_end)
2.    return;
3.  if (buf + off < buf)
4.    return;
```

The check at line 3 is simplified into false due to pointer overflow

```
model: |                                    # possible optimization
  %cmp3 = icmp ult i8* %add.ptr2, %buf
  -->  false
stack:                                      # bug location
  - buf.c:3
core:                                       # undefined behavior involved
  - buf.c:3
    - pointer overflow
```

# Evaluation

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

# STACK finds 160+ new bugs

- Applied STACK to many popular software
- Inspected warnings and submitted patches to developers
- Developers accepted most of our patches


iOS


Android


Chrome


Python

# STACK warnings are precise

Manually classify warnings and confirm with developers

- Kerberos: STACK produced 11 warnings
  - Developers accepted every patch (no warnings afterwards)
  - 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
- Positive user feedback

# 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 |

# Unstable code in the large: more bugs hiding

- 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

# Discussion: future compilers and languages

Lesson: undefined behavior ⇒ unstable code

- Compiler structures: better control for programmers
  - STACK: unified way of exploiting undefined behavior
  - Easier to turn on/off optimizations
  - Less sensitive to pass order
- Systems programming languages
  - Less undefined: (1 << 31) defined in next C++
  - More primitives: clang's __builtin_*_overflow, Rust
  - Performance trade-off: buffer overflow, race

# Summary

- Unstable code: a new species of bugs
  - Subtle
  - Significant security implications
  - SAT formulation and a practical tool STACK
- Language designers: be cautious about undefined behavior
- Compiler writers: use our techniques to generate better warnings
- Programmers: check your C/C++ code using STACK

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