IntFlow: Integer Error Handling With Information Flow Tracking

Marios Pomonis Theofilos Petsios
Kangkook Jee Michalis Polychronakis
Angelos D. Keromytis

December 7, 2014

Columbia University

Integer Error

Example

```
img_t *table_ptr;
unsigned int num_imgs = get_num_imgs();
unsigned int alloc_size = sizeof(img_t) * num_imgs;
table_ptr = (img_t *) malloc(alloc_size);
for (i = 0; i < num_imgs; i++)
    table_ptr[i] = read_img(i);
```

Integer Errors

- Mathematical representation vs machine representation
- Instances:
  - Integer overflow/underflow
  - Precision loss
  - Signedness change

Characteristics

- Mainly C/C++ specific:
  - Signed integers only (Java, Python)
  - Overflow protection (Python)
- Undefined:
  - Negative $\rightarrow$ unsigned
  - INT_MAX + 1
  - Optimizations
  - Expected behavior

Importance

- Can lead to buffer overflows, memory leaks etc...
  - Integral part of exploits
  - Erroneous memory allocation
- Integer overflow in top 25 most dangerous software errors
- $>$ 50 vulnerability reports (CVE) in 2014
  - QuickTime $\rightarrow$ Signedness change
  - launchd (iOS) $\rightarrow$ Integer overflow
  - Wireshark $\rightarrow$ Signedness change
  - Google Chrome $\rightarrow$ Integer overflow

Integer Overflow Checker (IOC)[ICSE2012]

- Clang AST
- Dangerous operation
  - Static: operation → safe function
  - Dynamic: detect errors
  - Report and (optionally) abort
- Clang trunk v3.3

```
/* a = b + c */
bool error = false;
a = safe_add(b, c, error);
if (error)
    report();
```

Integer Overflow Checker (IOC)[ICSE2012]

- Dynamic detection mechanism
- Offline use
- Input set from user

IOC Issue

- Overly comprehensive
- Lack of severity level
- Error $\neq$ vulnerability

Developer Intended Violations

- Idioms → errors
- Controlled
  - Expected bahavior
  - Not affected by attacker
- IOC → report all
  - Large list
  - Manually distill **critical** errors

Examples

```
umax = (unsigned) -1;
neg = (char) INT_MAX;
smax = 1 << (WIDTH - 1) - 1;
smax++;
```

Intflow

Goals:

1. Eliminate reports of developer intended violations
2. Retain and highlight critical error reports

IntFlow

Challenges:

1. Can we identify potential vulnerabilities?
2. Can we identify potentially exploitable vulnerabilities?
3. Can we do it accurately?

Critical Arithmetic Errors

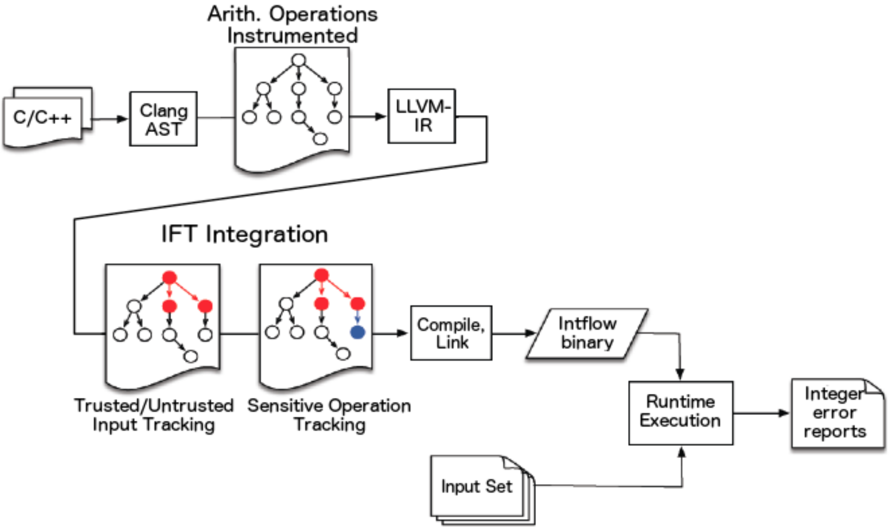
An error is potentially **critical** if:

1. **Untrusted source** → arithmetic error
   e.g. read(), getenv()...

OR

2. Arithmetic error → **sensitive sink**
   e.g. *alloc(), strcpy()...

IntFlow: Architecture



Static Information Flow Tracking

- Set of techniques analyzing data-flow
- Common compiler methodology
- Distinguishes flows to/from integer operations

Pros

✓ No runtime overhead

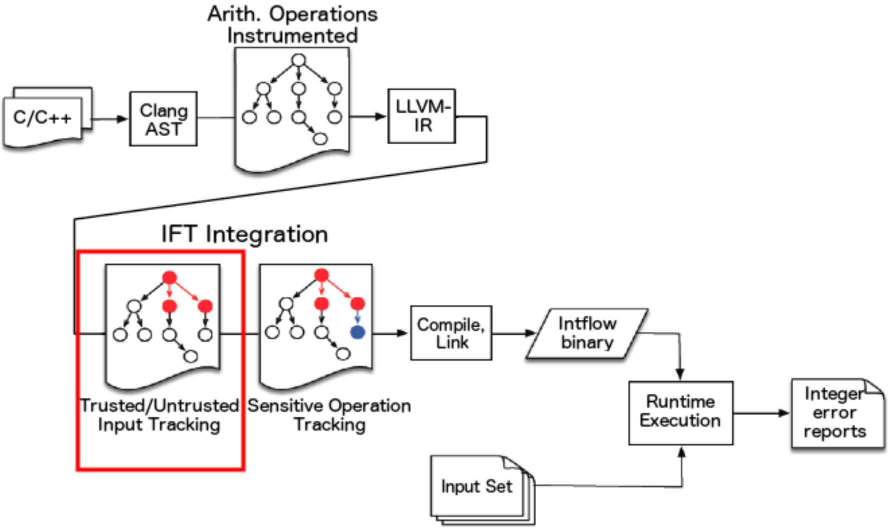
✓ Coverage

Cons

✗ Accuracy

✗ Scalability

IntFlow: Architecture



Backward Slicing: Operation → Sources

c = read();

b = gettimeofday();

b = 1;

c = -5;

```
a = safe_add(b, c, error);
if (error)
        report();
```

Forward Slicing: Source → Operation

```
b = gettimeofday();

return b;

send(b);

a = safe_add(b, c, error);
if (error)
        report();
```

Forward Slicing: Source → Operation

c = read();

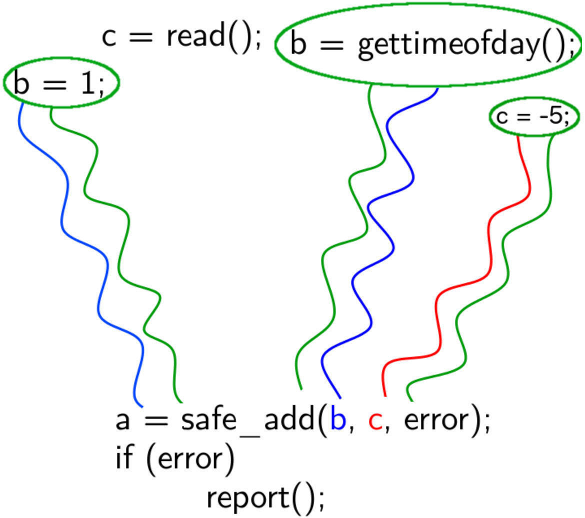
return c;

write(c);

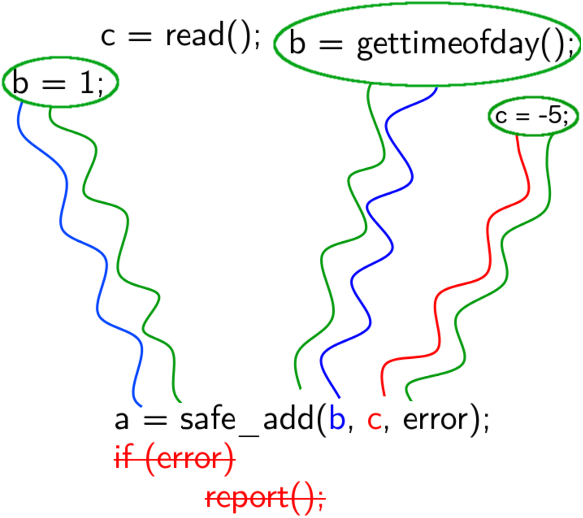
```
a = safe_add(b, c, error);
if (error)
        report();
```

Sources Examination

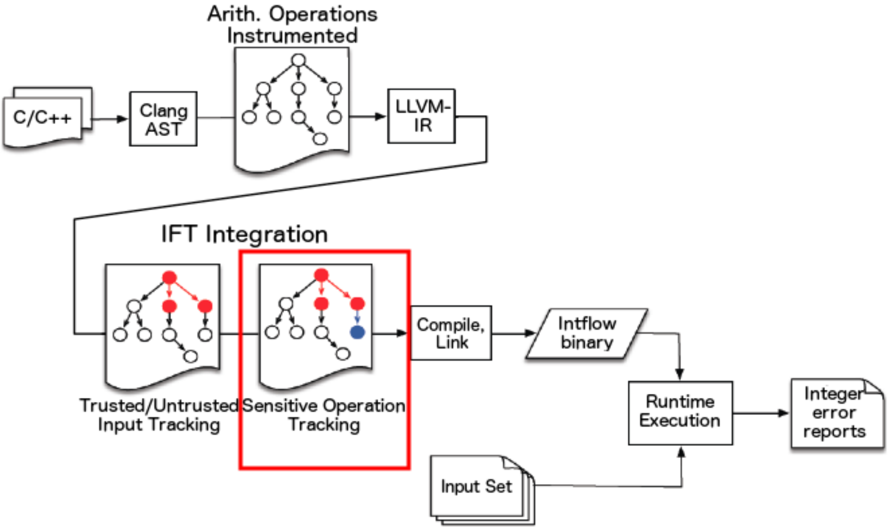
If sources = trusted → result = developer intended



Remove IOC Check



IntFlow: Architecture



Sensitive Operations

- Dynamic detection
- Operations → sensitive functions
- Operation → bit
- Check before a sensitive function
- Report if any bit set

```
...
1: x--;
...
2: x = y * z;
...
3: a++;
x *= a;
...
4: x++;
...
check_flags();
g = malloc(x);
```

Id	Error
1	T
2	F
3	T
4	F

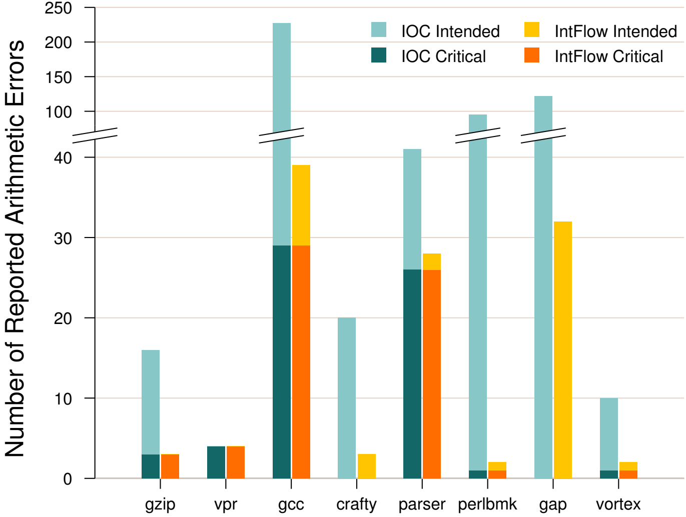
Modes Of Operation

- Blacklisting mode
  - **Untrusted sources** $\rightarrow$ operation
- Whitelisting mode
  - **Trusted sources** $\rightarrow$ operation
- Sensitive mode
  - Operation $\rightarrow$ **sensitive sinks**
- Combination of modes
  - Blacklisting/Whitelisting + Sensitive
  - $\uparrow$ Confidence - $\downarrow$ Completeness

Evaluation

- Whitelisting mode
  - Flexible
  - Context agnostic
    - ✓ Untrusted sources
    - ✓ Error propagation
  - Upper bound on report number

SPEC CINT2000



Real-world Applications

- Detected vulnerabilities:

CVE Number	Application	Error Type
CVE-2009-3481	Dillo	Integer Overflow
CVE-2012-3481	GIMP	Integer Overflow
CVE-2010-1516	Swftools	Integer Overflow
CVE-2013-6489	Pidgin	Signedness Change

- Produced reports

	Overall	Dillo	GIMP	Swftools	Pidgin
IOC	330	31	231	68	0
IntFlow	82	26	13	43	0

Runtime Overhead

- **Offline** use
- CPU-bound (e.g. grep): 50-80%
- IO-bound (e.g. nginx): 20%

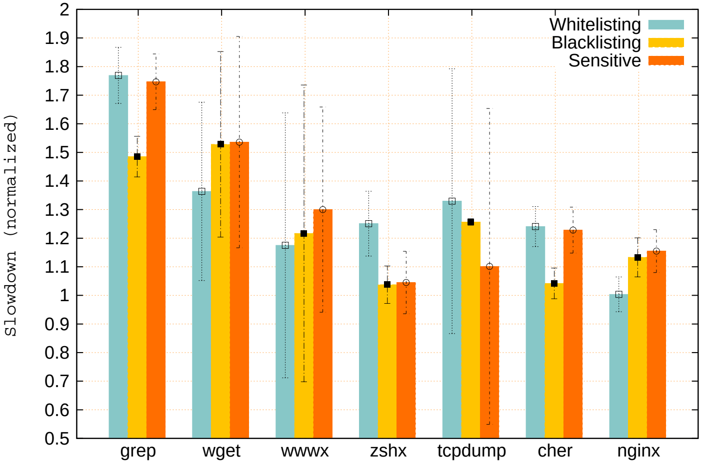
Summary

- Coupled IFT with IOC
- Identified critical errors
- Focused on potentially exploitable vulnerabilities
- Code: http://nsl.cs.columbia.edu/projects/intflow

Bonus

Backup Slides

Runtime Overhead



Additional Evaluation Results

- Independent stress test (red team)
  - Artificial vulnerabilities in popular applications
  - IO Inputs
    - Good: no exploit $\rightarrow$ normal execution
    - Bad: exploit $\rightarrow$ detect and abort
  - Aggregate result ($\frac{TP+TN}{Total}$): 79.30%