Instrumented Fuzz Testing using AIR Integers

Will Dormann [wd@cert.org]
Robert Seacord [rcs@cert.org]

Agenda

AIR Integers

Implementation

Performance

Fuzz Testing

Experiment

Future Work

Summary

As-If Infinitely Ranged Integers

The purpose of the AIR integer model is to either

- produce a value which is equivalent to a value that would have been obtained using infinitely ranged integers
- result in a runtime constraint violation.

This model is generally applied to both signed and unsigned integers but may be enabled or disabled per compilation unit.

AIR Integer Model

In the AIR integer model, when an observation point is reached, and before any critical undefined behavior occurs, if traps have not been disabled, and if no traps have been raised, then any integer value in the output is correctly represented ("as if infinitely ranged").

An observation point occurs at an output, including a volatile object access.

Traps are implemented using either

- existing hardware traps (such as divide-by-zero)
- by invoking a runtime-constraint handler

Observation Points

AIR Integers do not require where an exception is raised every time there is an integer overflow or truncation error.

It is acceptable to delay catching an incorrectly represented value until an observation point is reached just before it either

- affects the output
- causes *a* critical undefined behavior (as defined by the C1X Analyzability Annex).

This model improves the ability of compilers to optimize, without sacrificing safety and security.

Configuration

Requirements

- A patched version of GCC 4.4.0 or GCC 4.5.0 to insert the overflow and truncation checks
- A patched `stdlib.h` file to include the runtime-constraint handler definitions from ISO/IEC TR 24731-1
- The `libconstraint` library, which defines the constraint handlers used by AIR Integers

All of this is available from

http://www.cert.org/secure-coding/integralsecurity.html

Using AIR Integers 1

AIR Integers do not require changes to source code and can be used with pre-existing systems.

After configuring a system properly, AIR Integers can be used simply by recompiling a program with the following flags:

```
gcc -std=c99 -D__STDC_WANT_LIB_EXT__
  -fcheck-overflow=n -fcheck-truncation
  -lconstraint -o prog prog.c
```

Using AIR Integers 2

`-D__STDC_WANT_LIB_EXT__` enables the constraint handler definitions in `stdlib.h`

`-fcheck-overflow=n` enables overflow checking

- n=0 – Disable overflow checks (default)
- n=1 – Enable signed overflow checks
- n=2 – Enable signed/unsigned overflow checks

`-fcheck-truncation` enables truncation checks

`-lconstraint` links the constraint handler library

Constraint Handling

Constraint handlers are defined in the C1X normative Annex K “Bounds-checking interfaces” and by ISO/IEC TR 24731-1 to allow functions to trap when their runtime constraints are violated.

Constraint handlers have the following type.

```
typedef void (*constraint_handler_t)(
  const char * restrict msg,
  void * restrict ptr,
  errno_t error
);
```

Constraint Handling 1

AIR Integers uses the custom `libconstraint` library to define and implement simple constraint handlers to be called when overflow or truncation occurs.

- `abort_handler_s` : Indicate error on `stderr` and abort
- `ignore_handler_s` : Continue execution as normal
- `notify_handler_s` : Indicate error on `stderr` and continue

Constraint Handling 2

The default handler is `notify_handler_s`, which prints errors like

```
*** Runtime constraint violated:
    Signed integer overflow in addition at address 0x806dc4e
```

The TR 24731-1 defined function `set_constraint_handler_s`, can be used to register a different handler at runtime.

Custom constraint handlers can also be defined and registered at runtime.

Agenda

AIR Integers

Implementation

Performance

Fuzz Testing

Experiment

Future Work

Summary

Testing vs. Runtime Protection

AIR integers can be used in both dynamic analysis and as a runtime protection scheme.

There is a well understood tradeoff between runtime overhead and development costs.

- Providing correctness “guarantees” requires extensive testing and excruciating attention to detail
- Development costs can be decreased by adding runtime protection mechanisms however this will
  - increase the size of the executable
  - Introduce runtime overhead
- Runtime protection mechanisms still require a viable recovery strategy
- It is reasonable to provide some level of assurance combined with runtime checks, but you don’t want to pay twice

Benchmark Tests

To evaluate the performance of AIR Integers, runs of the integer portion of the SPEC CPU2006 benchmark were performed.

This benchmark evaluates compilers by compiling and running multiple packages, such as bzip2, gcc, and libquantum.

Performance Results

These ratios summarize the size of the runtime penalty imposed by using AIR Integers.

The ratios are measured against a consistent standard. Higher ratios reflect better performance.

Optimization Level	Control Ratio	AIR Integer Ratio	Percent Overhead
O0	4.93	4.65	6.02
O1	7.28	6.90	5.51
O2	7.45	7.08	5.22

Performance Analysis

The benchmark test with AIR Integers was performed with full overflow and truncation checking enabled.

However, actual calls to constraint handlers were replaced by `nop` instructions (ideally, these should be `call` instructions because `nop` is shorter resulting in better code density).

This avoids confounding the performance overhead of the checks with the overhead of constraint handlers.

Agenda

AIR Integers

Implementation

Performance

Fuzz Testing

Experiment

Future Work

Summary

Smart (Generational) Fuzzing

Requirements:

- In-depth knowledge of fuzzing target
- Specialized tools (e.g. Dranzer)
- Smart People

Results:

- Less crash analysis required
- Little duplication in results

Dumb (Mutation) Fuzzing

Requirements:

- No knowledge of fuzzing target
- Existing tools available
- Anybody can run the fuzzers

Results:

- More crash analysis required
- Much duplication in results

Winner: Dumb Fuzzing

The Fuzzers

Tavis Ormandy’s “fuzz”

- http://freshmeat.net/projects/taviso-fuzz
- Runs various patterns of random mangling on a file
- Looks at return code of process
- Linux-only
- Cannot save state or be resumed
- Fragile
- File formats only
- Unobtrusive

Taviso-fuzz

Example syntax:

```
fuzz -T 1 -m 1:4 -d /mnt/hgfs/fuzz/tiff "ffmpeg -y
-i __FILE__ -acodec pcm_s16le -f rawvideo /dev/null"
smclock.ogv
```

`-T <n>`	Timeout (seconds)
`-m <x>:<y>`	Load distribution (x of y machines)
`-d <dir>`	Store crashing testcases in dir
`“program __FILE__”`	Fuzz target syntax, `__FILE__` is the fuzzed file
`smclock.ogv`	The “seed” file

The Fuzzers

Caca labs zzuf

- http://caca.zoy.org/wiki/zzuf
- Random, repeatable mangling of a file
- User-specified randomization percentage
- Linux and OS X supported
- Saves state and can be resumed
- Robust
- File formats, network
- Intrusive

zzuf

Example syntax:

```
zzuf -cS -s0:10000 -r0.00001:0.1 -t 1 ffmpeg -y -i
smclock.ogv -acodec pcm_s16le -f rawvideo /dev/null
```

`-c`	Only fuzz files that appear on command line
`-S`	Prevent installation of signal handlers
`-s<w>:<x>`	Fuzz with seed number range from `<w>` to `<x>`
`-r<y>:<z>`	Randomization range from `<y>` to `<z>`
`-t <timeout>`	Application timeout

Verification

Platform differences

- Adobe Reader on Linux vs. Windows

Crash details

- Taviso `fuzz` and `zzuf` don't use a debugger
- `cdb` / `!exploitable`, `gdb`, `valgrind`

Unique crash determination

- `unique.sh` – memory location of crash
- Last line of source code before crash
- Hash of multiple lines before crash

Caveats

Default Ubuntu VM is bad for fuzz testing

- Gnome has lots of memory leaks
- Gnome has lots of overhead
- By default, Ubuntu has memory randomization enabled

Solution:

- Use a lightweight window manager like `fvwm` or `fluxbox` and configure the window manager to not raise new windows
- Disable memory randomization in `/etc/sysctl.conf`

```
Kernel.randomize_va_space=0
```

Caveats

Non-optimized debug build required for reliable debugging

- `./configure –disable-optimization –enable-debug`
- In Makefile:
  - `STRIP = /bin/true`
  - Remove any `–O2` or other optimization flags

Caveat Caveat

- Non-optimized code doesn’t always crash like optimized code

Fuzzing Variables

Fuzzing effectiveness depends on many variables:

- Fuzzer
- Mutation strategy
- Seed File
  - Program used to generate
  - Options used for generation
  - Size

Agenda

AIR Integers

Implementation

Performance

Fuzz Testing

Experiment

Future Work

Summary

AIR Analysis Techniques

Just run the code

- During normal operation of an application, integer constraint violations may be reported

Look at crashing test cases

- AIR constraint violations may be present in test cases that cause an application to crash: Correlation != Causation

Look at all fuzzed mutations

- AIR may report integer constraint violations that do not necessarily lead to crashes
- Lots of duplicate violations, e.g. 500 fuzzed variants / sec.

Experiment

AIR Integers have been used successfully to analyze two software libraries: JasPer and FFmpeg.

With the help of fuzzing tools, a number of overflows and truncations have been found.

Static analysis tools (such as splint) have been used by several classes of CMU graduate and undergraduate students to discover integer defects not detected by AIR integers.

False Positives

Instrumented fuzz testing all raised a number of false positives.

False positives are traps for overflows or truncations that are not errors because they are harmless for that particular implementation.

CERT C Secure Integer Guidelines

INT30-C. Ensure that unsigned integer operations do not wrap

INT31-C. Ensure that integer conversions do not result in lost or misinterpreted data*

INT32-C. Ensure that operations on signed integers do not result in overflow

INT34-C. Do not shift a negative number of bits or more bits than exist in the operand

INT35-C. Evaluate integer expressions in a larger size before comparing or assigning to that size

* No truncation errors were included in the results being presented today because of a defect in the prototype.

JasPer JPEG 2000 Project

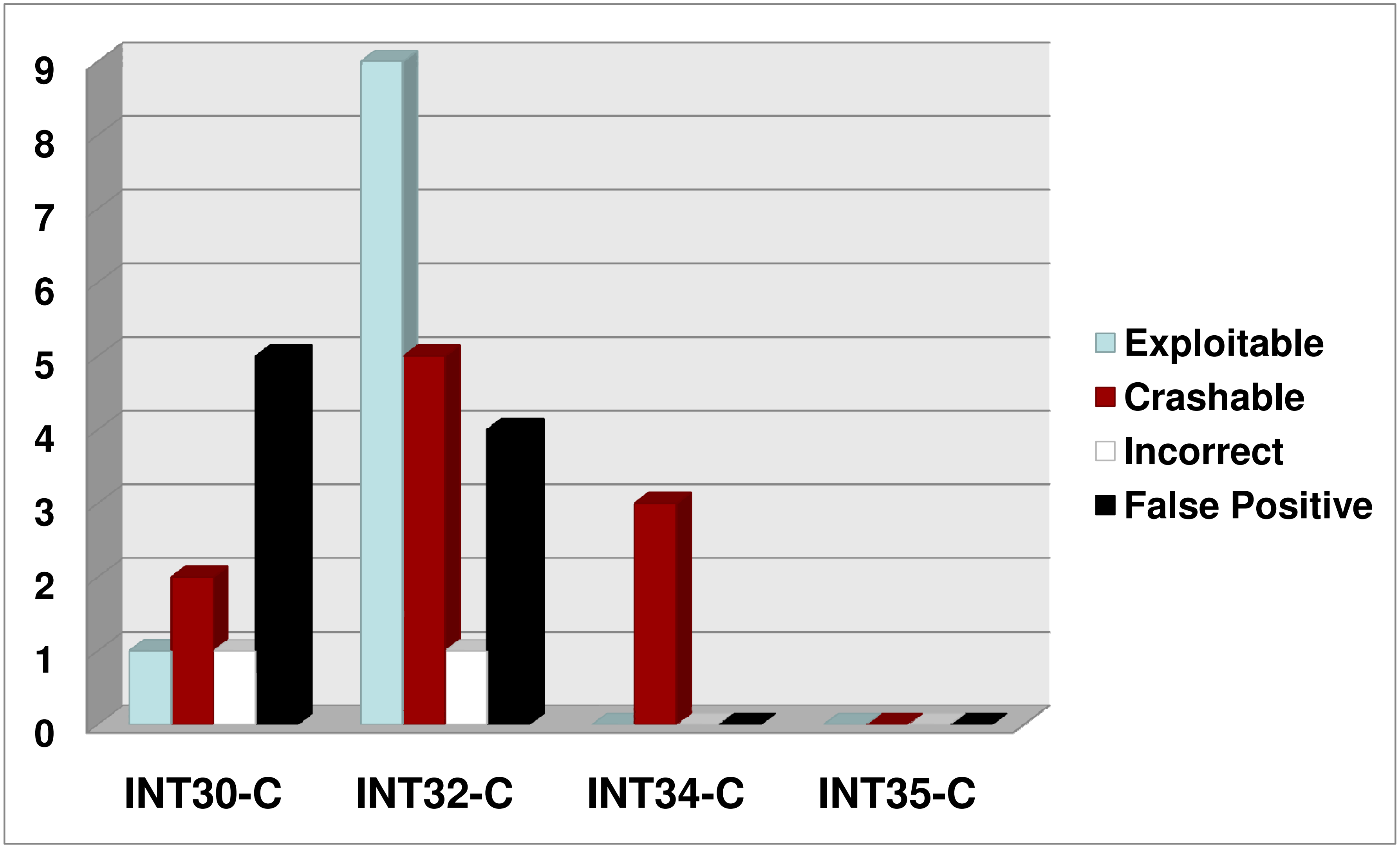
JasPer is a popular software toolkit for the handling of JPEG 2000 image data.

JasPer can be used to manipulate image data as well as import/export images in a variety of formats.

Several integer overflows and truncations have been detected in JasPer by using AIR Integers in combination with fuzzing tools.

Used by: KDE, ImageMagick, Ghostscript and more

JasPer Diagnostics



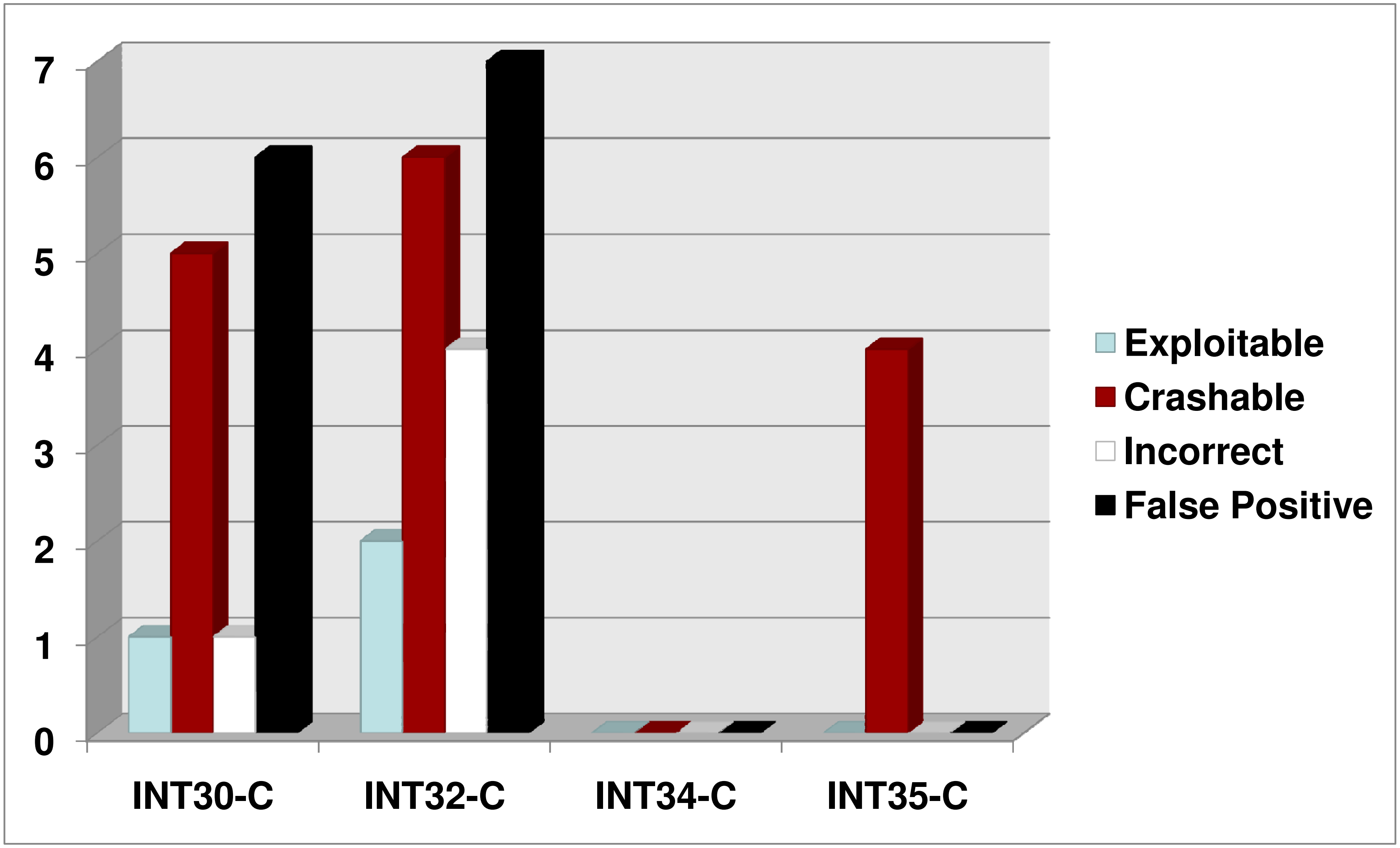
FFmpeg

FFmpeg is a popular tool for recording, converting, and streaming audio and video.

Many projects use code from FFmpeg, such as mplayer, VLC, Handbrake, Google Chrome, and ffdshow.

Combining fuzzing tools with AIR Integers revealed many integer overflows and truncations in FFmpeg.

Ffmpeg Diagnostics



Agenda

Usage

Implementation

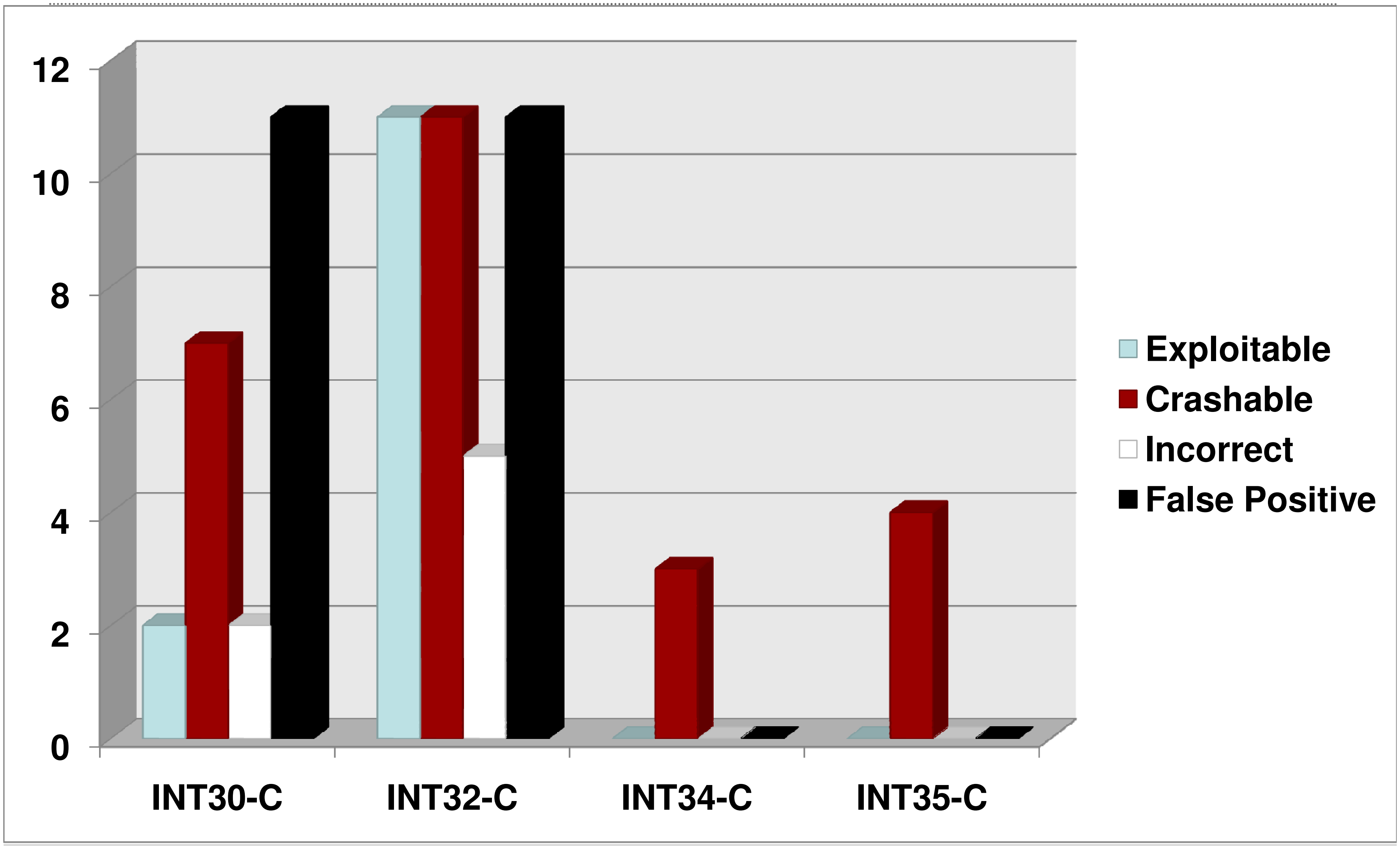
Performance

Results

Future Work

Summary

Jasper and FFmpeg Combined Diagnostics



Summary

Instrumented fuzz testing with AIR integers has some false positives resulting from nonconforming coding practices.

Code can be refactored to eliminate diagnostics

False negative rate (as measured using static analysis tools) surprisingly low.

Runtime overhead of AIR integers is low (and can be made lower) so retaining runtime protection is a viable option.

For More Information

Visit CERT® web sites:

http://www.cert.org/vuls/discovery/

http://www.cert.org/secure-coding/

Contact Presenter

Will Dormann
wd@cert.org
(412) 268-8922

Robert C. Seacord
rcs@cert.org
(412) 268-7608

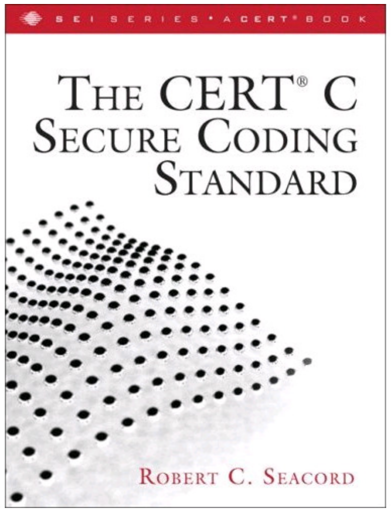
Contact CERT:

Software Engineering Institute

Carnegie Mellon University

4500 Fifth Avenue

Pittsburgh PA 15213-3890



SEI SERIES • A CERT® BOOK

THE CERT® C SECURE CODING STANDARD

ROBERT C. SEACORD

Acknowledgments

We got a lot of help:

Roger Dannenberg, David Keaton, Thomas Plum,

David Svoboda, Alex Volkovitsky, Timothy Wilson