Towards Vulnerability Discovery Using Staged Program Analysis

Bhargava Shastry[1], Fabian Yamaguchi[2],
Konrad Rieck[2], Jean-Pierre Seifert[1]

[1]Security in Telecommunications, TU Berlin
[2]Institute of System Security, TU Braunschweig

Institute of System Security

Introduction

- Fixing **vulnerabilities** early in systems code **still relevant**
- Critical infrastructure, mass deployed embedded systems run C/C++ code
  - Memory corruption vulnerabilities
  - Variety of attacks: ROP, heap-spray

But, hasn't static analysis been tried-and-tested?

- Yes, and no
- Yes, because
  - Frameworks like ITS4 to Coverity today use static analysis to find vulnerabilities
- No, because
  - **C++** static analysis is relatively **new**
  - How to deal with
    - **dynamic** language features?
    - novel programming paradigms e.g. **object-oriented programming**
  - Bug reporting is crucial yet underappreciated
    - Bug reported but unpatched is still a bug

SECT

Institute of System Security

Why is C++ a big deal?

```
class foo {
 public :
        int x ;
        foo () {} // Constructor doesn't initialize "x"
        bool isZero();
};
```

```
# include "foo.h"

bool foo :: isZero () {
        if ( !x )        // Potentially uninitialized
            return true ;
}
```

```
# include "foo.h"

int main () {
        foo f;  // Calls constructor (in header)
        if ( f.isZero() ) // Calls method (in source)
            return 0;
        return 1;
}
```

Bug manifests **across** source file boundary

Institute of System Security

Problem

Current open-source tools don't flag bugs spanning source boundaries

- Maybe, this isn't as big an issue? **Wrong!**
- Majority of **Chromium**, **Firefox** bugs span file boundaries
- Same is true for other large codebases e.g. **MySQL**

SECT

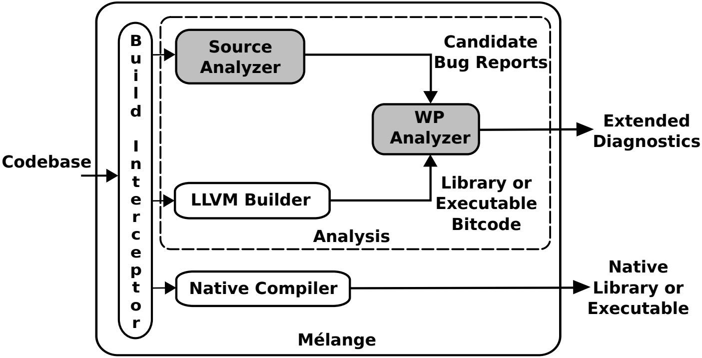
Institute of System Security

Our proposal

Melange

- Tackle multi-source-bugs by splitting analysis into **two stages**
- Stage 1: Analyze individual source files building up list of potential bugs
- Stage 2: Validate findings of stage 1 by doing whole-program analysis

Overview



Codebase
Build Interceptor
Source Analyzer
Candidate Bug Reports
WP Analyzer
Extended Diagnostics
LLVM Builder
Library or Executable Bitcode
Analysis
Native Compiler
Native Library or Executable
Mélange

How it works (1/2)

- Analyze object implementations one-by-one

```
# include "foo.h"

bool foo :: isZero () {
        if ( !x )        // Potentially uninitialized
            return true ;
}
```

- Analysis happens alongside native compilation
- Flag potential bugs → *foo* :: *x* may be used uninitialized

How it works (2/2)

- Validate list of potential bugs

```
# include "foo.h"

int main () {
        foo f;   // Calls constructor (in header)
        if ( f.isZero() ) // Calls method (in source)
            return 0;
        return 1;
}
```

- Analysis happens post compilation and program linking
- Output a bug report

Bug Report

```
// Source-level bug report
// report-e6ed9c.html
...
Local Path to Bug: foo::x->_ZN3foo6isZeroEv

Annotated Source Code
foo.cpp:4:6: warning: Potentially uninitialized object field
 if (!x)
      ^
1 warning generated.

// Whole-program bug report
------------------- report-e6ed9c.html -------------------
[+] Parsing bug report report-e6ed9c.html
[+] Writing queries into LLVM pass header file
[+] Recompiling LLVM pass
[+] Running LLVM BugReportAnalyzer pass against main
-----------------------------------------------------------
Candidate callchain is:

foo::isZero()
main
-----------------------------------
```

Extensible

Analysis can be extended to **multiple bug classes**

- Prototype supports the following bug classes
  - Type confusion
  - Garbage reads
  - Sign extension/conversion
- Adding support for a bug class entails
  - Clang Static Analyzer plug-in, AND
  - LLVM optimizer plug-in
- Analysis complexity orders of magnitude lesser than analyzed programs
  - Melange spans $\approx$ 2.6 thousand LoC
  - Largest analysis target $\approx$ 14 million LoC

Evaluation

- **Effort** required to use
- **Benchmark** results
- **Effectiveness** in finding bugs in large codebases
- **Runtime** results

Usability

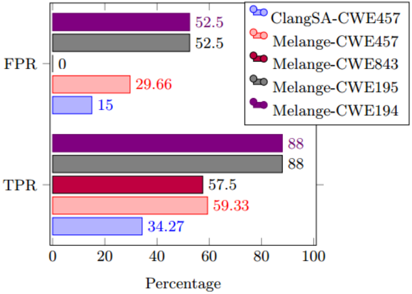
Melange is easy-to-use

- Package with a production compiler toolchaing (Clang/LLVM)
- Our analysis plugs into standard compiler
- This means running our analysis is a matter of adding a few extra flags
  - No knowledge of build system required
  - Analysis invocation transparent to user (developer)

SECT

Institute of System Security

Benchmark results



- Our analysis can be applied to multiple bug classes
- We have higher true positive rates compared to baseline
- Overall false positive rate is also higher but manageable
  - Security analyst/Developer can wade through them without being overwhelmed

SECT

Institute of System Security

Controlled evaluation

- Going through PHP bug reports, type confusion seems to be widespread
- We wrote a type checking Melange plugin and found five known exploitable vulnerabilities

Codebase	CVE ID	Bug ID
PHP	CVE-2015-4147	69085
PHP	CVE-2015-4148	69085
PHP	CVE-2014-3515	67492
PHP	Unassigned	73245
PHP	Unassigned	69152

Institute of System Security

Uncontrolled evaluation

- We analyzed Chromium, Firefox, MySQL releases from late 2015
- We found 3 bugs out of which 2 were rediscovered (previously found by fuzzing+dynamic analysis)
- Consistently found a handful of interesting potential bugs

Codebase (MLoC)	Bug reports		True positives
	Stage 1	Stage 2	
Chromium (14)	2686	12	2
Firefox (5)	587	16	1

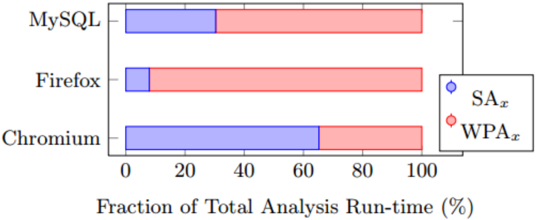
SECT

Institute of System Security

Performance (1/2)

Our analysis is **much slower** than native compilation time...

- Total analysis time varies between 30-45x compilation time
- Some codebases are particularly suited for staged analysis
  - Modular build system permits incremental analysis



Performance (2/2)

…But, it's **fast enough in practice**

- We rented an EC2 compute VM at $\approx$ 2 Euros/hour
- Total analysis runtime $\approx$ 48 hours $\approx$ **100 Euros**[1]
  - Firefox $\approx$ 31 hours $\approx$ 62 Euros
  - Chromium $\approx$ 13 hours $\approx$ 26 Euros
  - MySQL $\approx$ 4 hours $\approx$ 8 Euros
- Ours is a research prototype $\rightarrow$ Lots of room for optimizations

[1] For first analysis only. Incremental analyses are cheaper

Take aways

- Modern programming paradigms benefit from **staged analysis**
- Static analysis is **viable**
- Tools such as Melange
    - **complement** existing program testing techniques
    - Help **find** and **fix** bugs early

Source code

- Melange checker source code at `https://github.com/bshastry/melange-checkers`
- Demo box at `https://github.com/bshastry/vagrant-pallang`

Acknowledgements

Thank you for your attention! **Questions?**

Our thanks to

- Colleagues at SecT esp. **Janis Danisevskis**
- **Daniel Defreez**, UC Davis
- Grants from these projects
  - **Enzevalos**, **NEMESYS**, **DEVIL**

Institute of System Security