

# Practical Considerations in Control-Flow Integrity Monitoring

Iavor Diatchki   Lee Pike   Levent Erkök

Galois Inc.

The Second International Workshop on Security Testing

# Control Flow Integrity (CFI)

- Abadi, Budiu, Erlingsson, and Ligatti, 2005.
- The execution path of an application matches expectations.
- CFI may be violated if:
  - the program's code is modified,
  - the program's data is modified,
  - our expectations are incompatible with the implementation.

- 1 Introduction
- 2 Exploring the Design Space
- 3 What Works, What Doesn't
- 4 Future Work and Conclusions

# Use Static Analysis To Determine Expectations

- The program may be analyzed at different levels:
  - Source code
  - Intermediate compiler output
  - Compiled binaries
  
- Example:
  - Compute program's call-graph.
  - At run-time, check that function calls match the call-graph.
  
- Violations indicate potential attacks.
  - (or inaccuracies in the analysis)

# More About the Call Graph

- The call-graph of a program is a directed graph:
  - vertices are function symbols,
  - there is an edge  $F \rightarrow G$ , if  $F$  may call  $G$  directly.
  
- A trace of the calls in a program should correspond to a path in the call graph.
  
- A non-conforming trace will contain either:
  - an unknown vertex (i.e., a call from an unknown location), or
  - an unknown edge (i.e., an unexpected call from one function to another).

# Call-Graph Variations

- Using call-sites instead of functions:
  - Nodes are call-sites (i.e., return addresses) instead of functions.
  - $c_1 \rightarrow c_2$  if we can reach  $c_1$  from  $c_2$  directly.
  - A bit more precise.
  - Direct correspondence to control stack.
  
- Call-site relation,  $R$ :
  - Relates functions to call-sites.
  - $R f c$  if call-site  $c$  may call  $f$ .
  - One call-site may call multiple functions with function pointers.

# Some Choices

- How do we perform validation?
  - By instrumenting the program, or
  - By using an external process (e.g., attach via ptrace).
  
- How much do we validate?
  - Entire control stack, or
  - just last edge?
  
- How do we associate return address to nodes in the call graph?

# Monitoring Transitions (some more choices)

- In functions prologues:
  - A call from an unexpected location prevents execution.
  - However, an attacker could skip the prologue.
  - Useful because it prevents unexpected nested calls.
  
- In function epilogues:
  - Difficult to avoid the check, but
  - by the time we detect the exploit the function has finished executing.
  - Useful because it detects problem, if a bit late.

# Implementations: part 1, similarities

- To explore the space we tried two implementations.
- Both implementations use instrumentation.
- In both cases we use flags to control:
  - where to insert checks (prologue vs. epilogue),
  - if we should check
    - the whole stack every time (slow!), or
    - just the last transition (fast!).
- (Checking the whole stack makes more sense for an external monitor.)

## Implementations: part 2, differences

- The “C” implementation:
  - Analyzes C source code.
  - Uses a conventional call-graph.
  - A hash-table maps call-sites to graph nodes.
  
- The “assembly” implementation:
  - Analyzes the assembly generated by compiler.
  - Uses a call-site relation.
  - Uses a linear lookup to validate caller.

1 Introduction

2 Exploring the Design Space

3 What Works, What Doesn't

4 Future Work and Conclusions

# Call Graphs Over-Approximate Execution

- Not every path through the CG corresponds to a valid execution trace:

```
void f() {  
    g(CMD1);  
    ...  
}
```

```
void g(int cmd) {  
    switch(cmd) {  
        case CMD1: h1(); break;  
        case CMD2: h2(); break;  
        ...  
    }  
}
```

- Just using a call-graph does not detect all CFI violations.

# Function Pointers

- Many programs use function pointers:
  - caller determined at run-time.
  
- We tried the following options:
  - 1 Ignore function pointers:
    - Results in missing edges in call-graph: false positives.
  
  - 2 Assume that *any* function may be called.
    - Too weak? It could miss important CFI violations.
  
  - 3 Use a data-flow analysis to improve on 2.

# Libraries

- We need call-graphs for libraries.
- Library call-graphs are “linked” together to compute entire graph.
- Missing library call-graphs result in incomplete call-graphs:
  - We know the symbols, so no missing nodes.
  - We don't know implementations: missing edges.
  - Monitoring may report false positives.

# Optimizations

- Some optimizations invalidate call-graphs computes from source.
- Consider:

$f \longrightarrow g \longrightarrow h$

- Inlining: call to  $g$  inlined in  $f$ .

$f \overset{\curvearrowright} {\longrightarrow} g \longrightarrow h$

- Tail call:  $g$  makes a tail-call to  $h$ .

$f \longrightarrow g \overset{\curvearrowright} {\longrightarrow} h$

# Improving the Call-Graph Through Dynamic Analysis

- We can improve the call-graph by “learning” in a safe environment.
- Instrument program for “learning”:
  - While executing, program records valid execution traces.
- Execute instrumented program on test data.
- Helps identify:
  - missing edges due to approximations in static analysis,
  - paths through the call-graph that are known to be valid (i.e., not over-approximations)

- 1 Introduction
- 2 Exploring the Design Space
- 3 What Works, What Doesn't
- 4 Future Work and Conclusions

# GCC Plugins

- gcc 4.5 supports user defined plug-ins.
- A much simpler way to get access to a (correct) call-graph.
- The new gcc supports link-time optimization (LTO).
  - Object-files compiled with LTO already contain call-graph.
- By using a gcc plugin we gain access to other useful information.

# Explicit Developer Expectations

- Call-graphs are not the only choice.
- Allow programmers to specify properties about control flow.
- For example, LTL-like properties:
  - Function  $F$  should never call  $G$ , transitively.
  - Function  $F$  should be called only by descendants of  $G$ .
  - Checks fit naturally with our framework.
- Violations indicate:
  - Potential attacks, or
  - implementation bugs, because developer's expectations were not met.

# Conclusions

- CFI monitoring makes it more difficult to attack applications.
- A CFI monitor can be quite simple.
- Instrumented programs have reasonable performance.
- There are multiple design choices:
  - The best choice may be system dependent.