Control-Flow Integrity

PRECISION, SECURITY, AND PERFORMANCE

ANDI TURDIU
Agenda

- Introduction to control flow hijacking
- Introduction to CFI
- Classification of Control-Flow transfers
- Points to analysis: A theoretical perspective
- Points to analysis: A practical perspective
- Evaluation:
  - Precision
  - Security
  - Performance
- Results
- Remarks
Introduction

• Low-level programming languages such as C and C++
  • high degree of freedom to optimize and control resources, leads to:
    ➢ Efficient programs
    ➢ Manual memory management and type rules
    ➢ But also to: security vulnerabilities.

Chief among these are buffer overflows and other types of memory corruption that are routinely exploited by attackers.

• Modern attacks hijack the control-flow of the application
  ❖ Reuse existing code

• E.g. Language enforces type safety:
  • Change memory location orthogonally to the guarantees and proofs:
    ➢ Hijacked control flow
Introduction
CFI

*CFI restricts the allowed transfers to a smaller set of targets*

- **CFI**: policy that restricts the execution flow of a program at runtime
  - Divided into two phases:
    - The analysis
    - The runtime instrumentation

- Security guarantees of CFI depend on the **CFG**
  - Which is constructed during the **analysis phase**.
  - This always results in over-approximation: E.g. type-based

- **Runtime instrumentation** validates target addresses *before* they are used in an indirect control-flow transfer
  - Detects modified code pointers
Control-flow transfers can broadly be separated into two categories:

- Forward transfers
- Backward transfers

- Forward transfers *moves* control to a new location
- Backward transfers *returns* control to a prior location

- Typical forward transfer instructions (ISA):
  - Call & jump: (direct jump, \textit{direct call}, \textit{indirect jump}, \textit{indirect call})
  - Backward transfers instructions (ISA):
    - Return

Control-flow transfers can become more complicated in the presence of more involved features:

- \textit{Exception handling}
Points to analysis
Theoretical perspective

*Analysis that identifies indirect calls/jumps targets*

Many compiler optimizations benefit from points-to analysis. As a result, points-to analysis must be sound at all times and therefore conservatively overapproximates results.

- We distinguish between:
  - Flow-sensitive vs. flow-insensitive
  - Context-sensitive vs. context-insensitive

*Context* and *flow* sensitivity are orthogonal and a points-to analysis combining both yields higher precision.
Points to analysis
Theoretical perspective:

Flow-sensitive vs flow-insensitive

A flow sensitive analysis considers the state of the program per line. A flow insensitive analysis computes sets that are valid for the whole program.

From a CFI perspective a flow-sensitive points-to analysis offers higher precision.
Points to analysis
Theoretical perspective:

*Context-sensitive vs Context-insensitive*

A **context-insensitive** analysis considers a function independent from its callers.
A **context-sensitive** uses additional context information to compute higher precision results.

*From a control-flow integrity perspective a context-sensitive points-to analysis offers higher precision.*
Points to analysis

Practical perspective:

• Points-to analysis over approximation:
  • reduces precision
  • restricts the optimization potential of programs.
• The reduced precision also reduces precision for CFI, opening the door for attackers:
  • More reachable targets = more vulnerable to control flow hijacks without violating the CFI policy.

Figure 2: Backward control-flow precision.
To evaluate different CFI mechanisms we use these axes:

- Supported directions of an approach (D)
- Supported control-flow transfers (CF)
- Reported performance numbers (RP)
- Static-analysis precision of forward control-flows (S.A.P.B)
- Static-analysis precision of backward control-flows (S.A.P.F)
- Quantitative argument (Q)
Quantitatively assessing how much security a CFI mechanism provides is challenging as attacks are often program dependent and different implementations might allow different attacks to succeed.

Known measurement: Average Indirect Target Reduction (AIR)
Unfortunately AIR is known to be a weak stand point.

E.G. 1.8MB of executable code with an AIR value of 99.9% still allows for 1841 targets, arguably enough for an arbitrary attack.
Security
Quantitative argument

A more meaningful metric must focus on the number of targets (number of equivalence classes) available to an attacker.

Smaller classes are more secure because they provide less attack surface.

Possible metric:

\[
\frac{1}{EC} \times LC = \text{QuantitativeSecurity}
\]

EC = nr. of equivalence classes
LC = size of the largest class
Performance

While the security properties of CFI have received most the attention, performance characteristics play a large part in determining which CFI mechanisms are likely to see adoption.

Surveys show:

- 10% overhead do not tend to see widespread
- overheads below 5% are desired by industry practitioners

As for all shown metrics, apples to apples comparison is difficult due to architecture, characteristics and implementation differences.

Test bench:
Dell PowerEdge T620 dual processor
Intel Xeon E5-2660 2.20GHz
64GB DDR3 Memory
64-bit Ubuntu Linux 14.04.2 LTS
Precision Results:
• $\pi$CFI offers the highest precision due to leveraging dynamic points-to information.

Security Results:
• $\pi$CFI reduces the equivalence classes available to the attacker by 21.6%

Performance results:
• Recent compiler based CFI mechanisms have mean overheads in the low single digit range
  - CFI enforcement is not too costly in practice
Remarks

These results are a cut down version of the Paper “Control-Flow Integrity: Precision, Security, and Performance” by Nathan Burow et.al.

They serve as a general guidance to the mentioned paper. For the details, differences in different CFI mechanisms and difficulties in creating a unified “test” please refer to the Paper.

It is a very interesting read and also easy to follow and understand.

Thank you very much for your attention!