Compiler Analysis Framework for Application-Aware Trust

Karthik Pattabiraman, William Healey, Shane Ryoo, Zbigniew Kalbarczyk, Ravi Iyer, Wen-mei Hwu

Error Detectors
- Identify instructions that compute the target variable at the detection location (from backward slice)
- Encode the instructions as a symbolic expression
- Reduce the expression to optimized logical expression
  - Only encode the variables that affect the value of the chosen variable at the detection location (program slicing)
  - Create specialized versions of the slice depending on the path that is followed at runtime (partial evaluation)
  - Instrument code to track paths followed at runtime
  - Choose check depending on path followed at runtime

Detector Placement
- Goal: Prevent error propagation in applications
- Fault Model: Errors in data values of program
- Both variable and program point chosen
- Analysis using Dynamic Dependence Graph
- Placed detectors according to heuristics e.g. Famouts, Lifetimes etc. computed from DDG
- 10 or more ideal detectors placed using fanouts can yield coverage of 80% (for gcc85) for manifested errors

Goal
- Derive detectors based on application-specific properties for reliability and security
- Detect both errors and attacks in the application
- Detectors to be derived automatically from source code using compiler analysis
- Extract properties of the application from source code semantics
- Enforce these properties at runtime using a combination of software and hardware (RSE)

Detector Derivation Example

Reliability
- Compute slicing of the program using backward slicing resulting from the critical variable and the program point at which it is checked
- Consider the critical variable the location within a program to place detectors for hard coverage

Security
- Use knowledge of the application semantics to place critical assertions across critical paths (e.g., program points)
- Static program analysis:
  - Compute backward slice of program variable along each program path starting from the critical variable and the program point at which it is checked
  - Check variable is used only by the path and is subject to the attacker

Hardware Implementation
- RSE is a reconfigurable processor-level framework for reliability and security
- Detectors implemented as RSE module consisting of
  - Shadow Register File: Holds the state of the checked location
  - Assertion Table: Stores the assertions parameters
  - Data path: check assertions independently from processor

Information -Flow Signatures
- Use detection of program data-flow violations as an indicator of malicious tampering with the system
- Prevent an attacker to exploit disconnected between source-level semantics and execution semantics of the program
- Security critical variables chosen based on app semantics
- Embed a compile-time static program analysis to
  - Extract a backward slice which contains all dependent instructions along each control path
  - Form a signature, which encodes dependencies as a set (or sequence) of instruction PCs along each control path
- Compute runtime signatures for each critical variable
  - Trusted bit associated with each instruction
  - Only trusted instructions can update runtime signatures
  - Check signatures for instructions with trusted bit set

Security Checking: How Do Signatures Detect Attacks?

Security Checking: Why do we need to encode entire dependency tree?