Application-Aware Security

Zbigniew Kalbarczyk

W. Healey, K. Pattabiraman, R. Iyer

Center for Reliability and High-Performance Computing

Coordinated Science Laboratory

University of Illinois, Urbana-Champaign

Goals

- **Detectors based on application-specific properties**
- **Early detection of attacks**
  - Detect attacks before they corrupt critical memory state
- **Derive detectors automatically from application**
  - Extract properties of the application using compiler analysis
  - Enforce extracted properties at runtime
- **Provide efficient hardware implementation**
  - Integration with Reliability and Security Engine (RSE)
  - Low-latency monitoring and detection at runtime

Derivation of Attack Detectors

- **Goal:** Preemptively protect "security-critical data"

- Technique: Information Flow Signatures
  - Derive dependences for critical variable using static analysis
  - Encode dependencies for critical variable(s) as signature
  - Check that the signature is not violated at runtime

- **Can be applied selectively to critical variables in program even though other variables are attacked**

- Threat/Attack Model
  - Memory Corruption Attacks (Buffer overflows, format string)
  - Hardware attacks (smart-card based attacks)
  - Insider attacks (malicious plugins, third party libraries)

Attack Detectors: Conceptual Example

```
1  int authenticate(char* username, char* password)
2  {
3  int authenticated=0;
4  int result;
5  char tmpbuf[512];
6
7  snprintf(tmpbuf,sizeof(tmpbuf),username);
8  tmpbuf[sizeof(tmpbuf)-1] = '\0';
9  syslog(LOG_NOTICE,tmpbuf);
10 authenticated |= result;
```

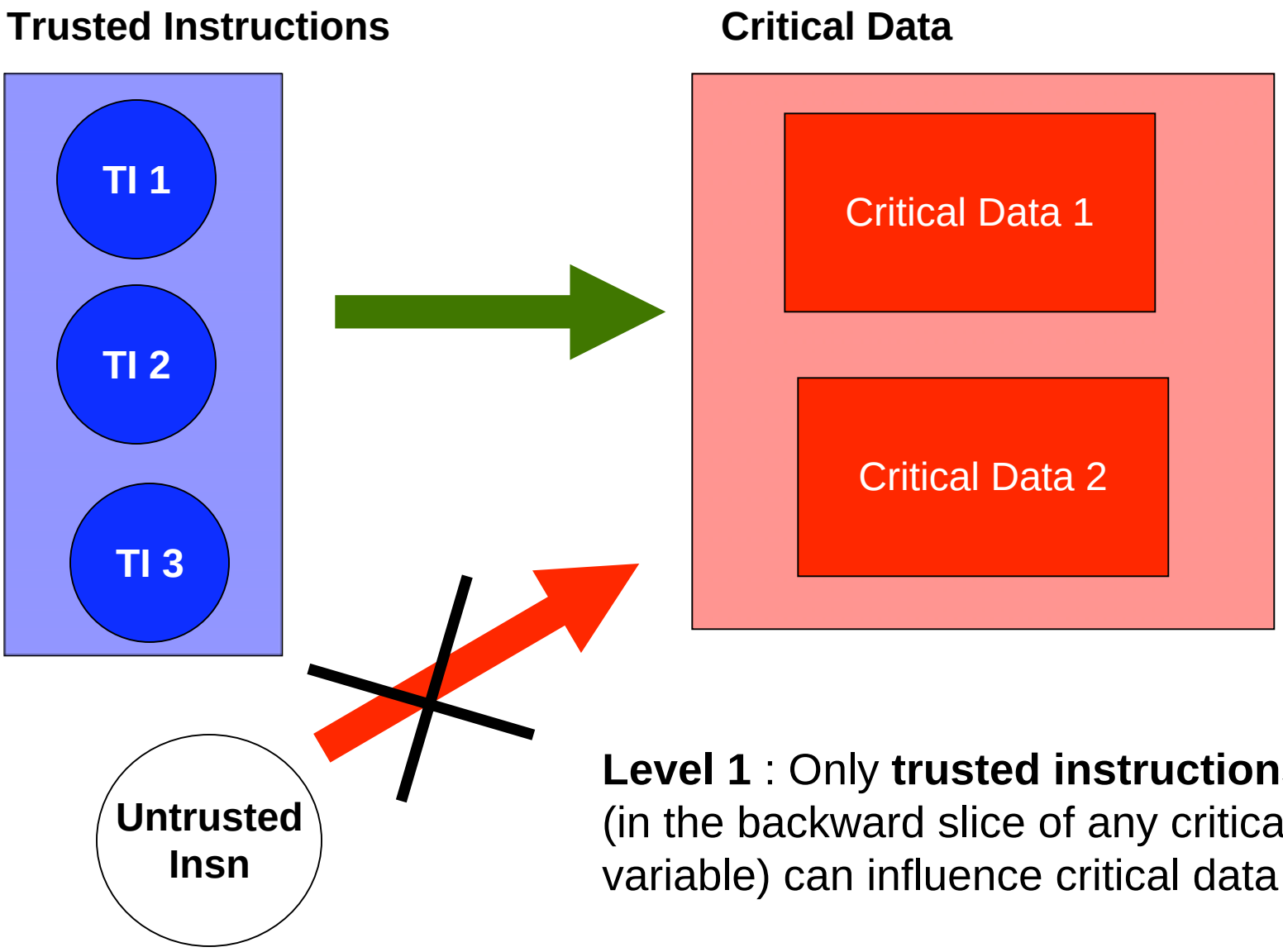
Attacker overwrites result instead, realizing that it influences authenticated

Attacker overwrites authenticated via format string

Information-flow signatures encode not just direct dependences, but also indirect dependences

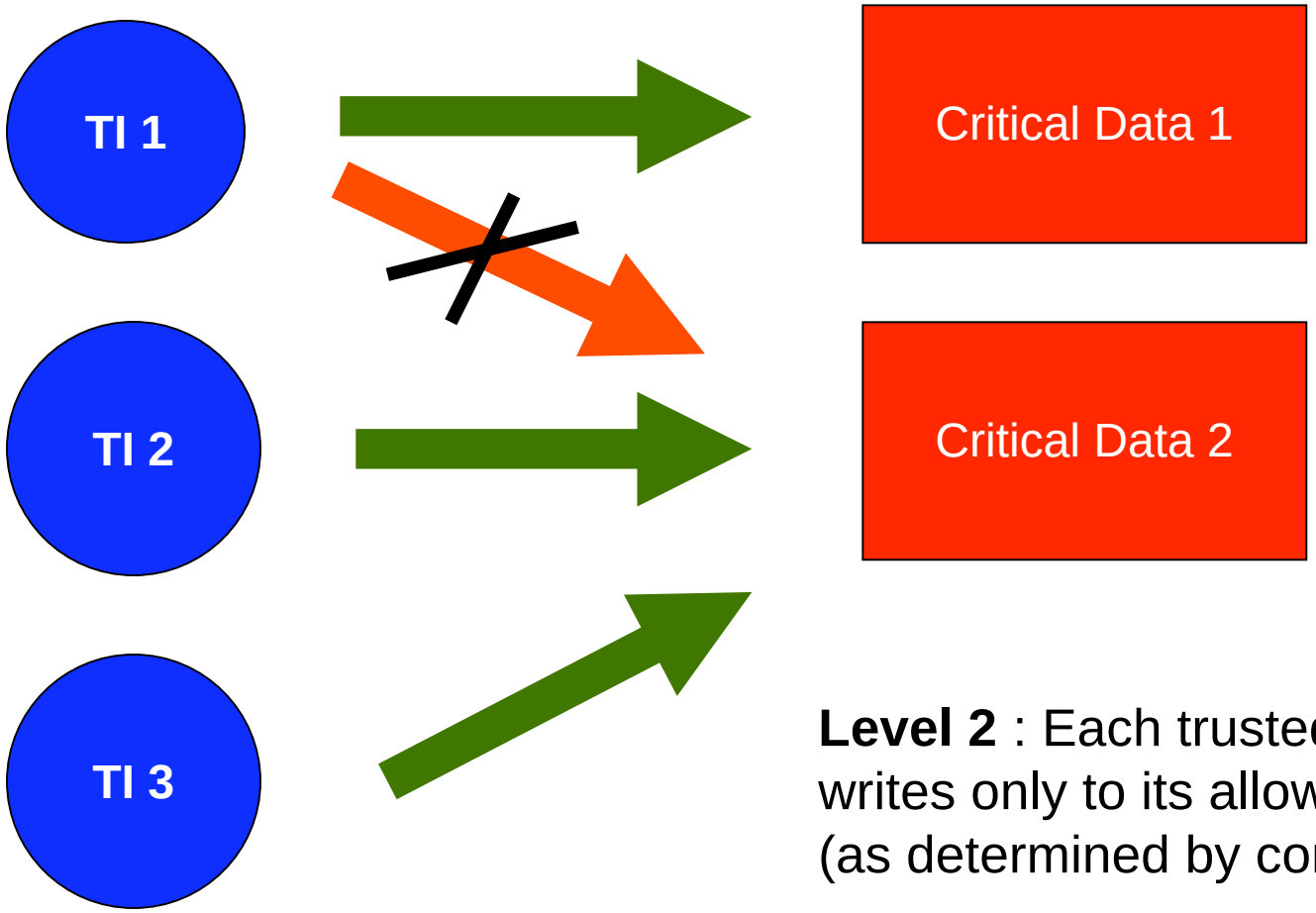
Critical Variable: authenticated; Signature: {10,6},{3}

Level 1 Check: Trustedness



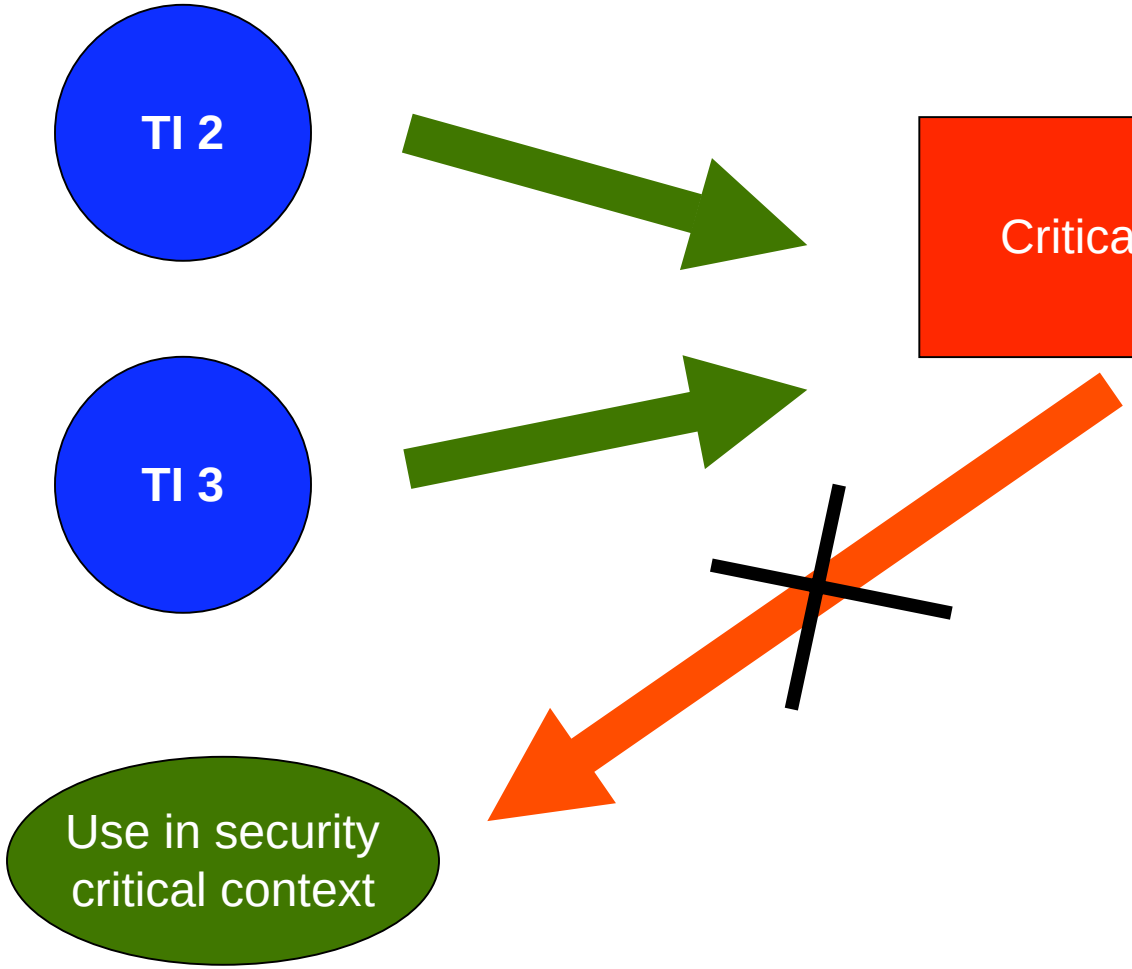
Level 1 : Only **trusted instructions** (in the backward slice of any critical variable) can influence critical data

Level 2 Check: Verification of Trust



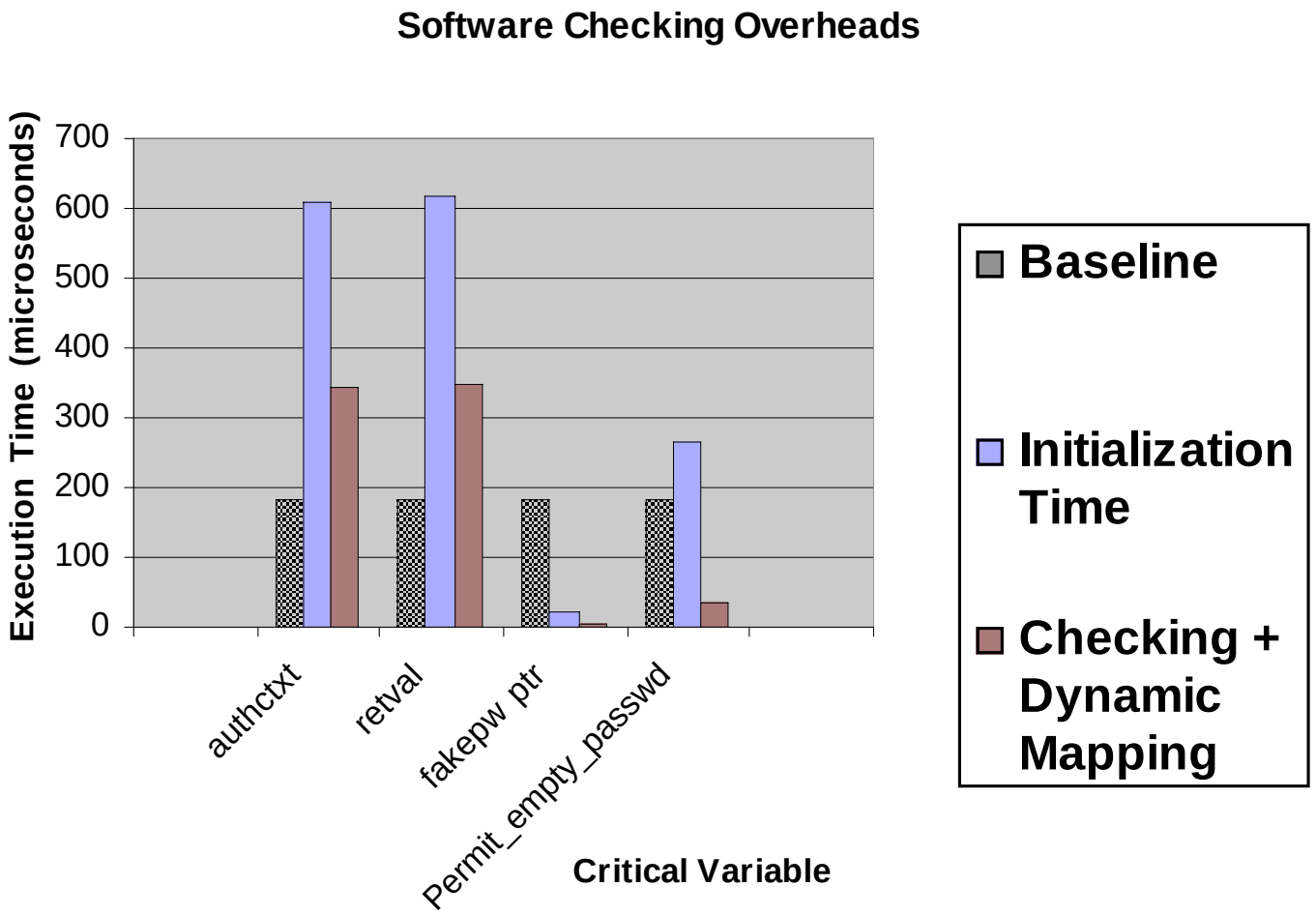
Level 2 : Each trusted instruction writes only to its allowed targets (as determined by compiler)

Level 3 Check: Completeness



Level 3 : All trusted instructions that write to critical data (along a control path), do so before its use in a security critical context

Example Results for Attack Detectors: OpenSHH

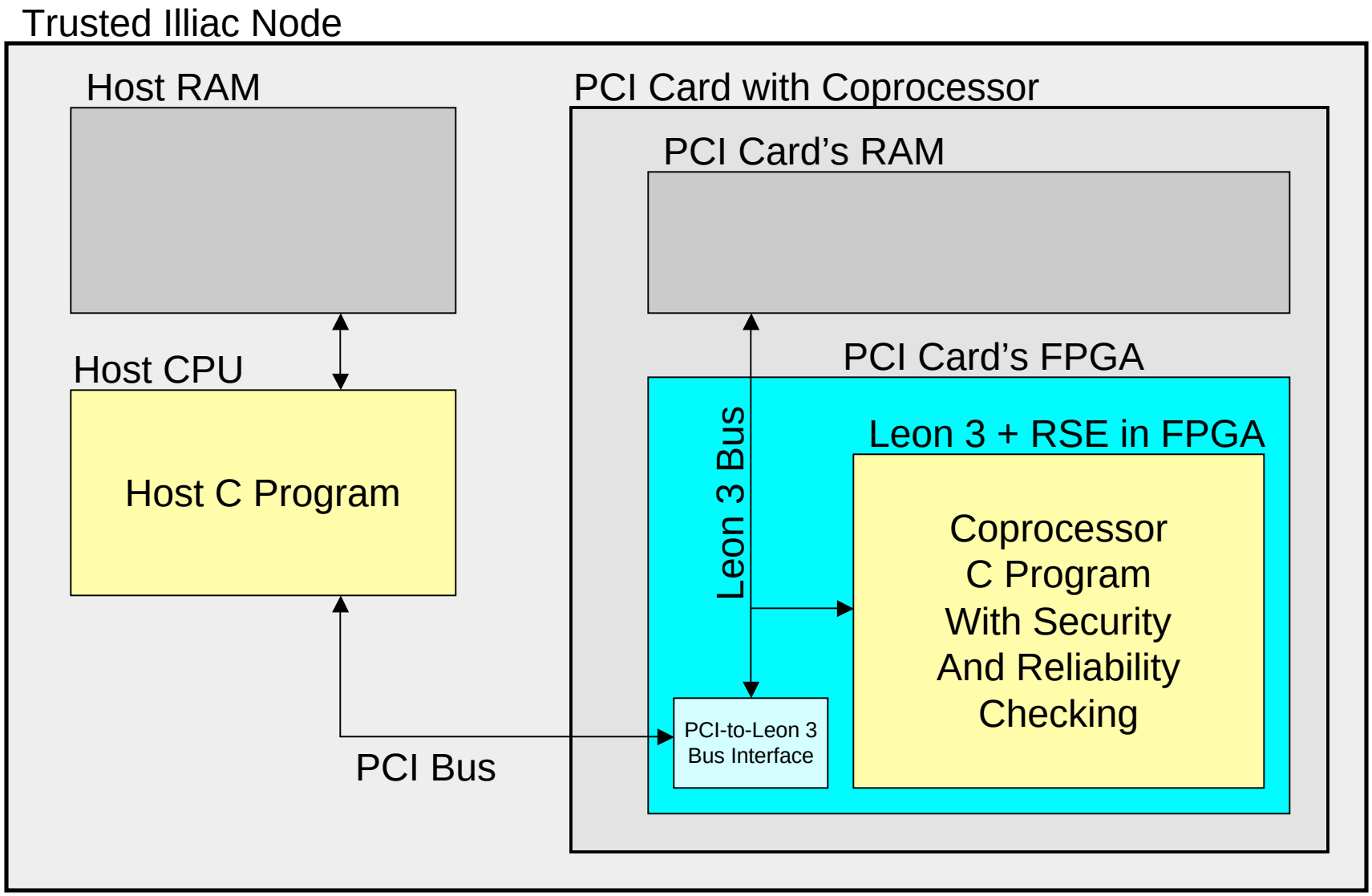


Performance overhead depends on length of dependence trees of critical variables and the size of the set of allowed targets for trusted instructions

Hardware Implmentation

- FPGA implementation and sythesis of the Level 1 and Level 3 checks
- Performance overhead:
  - 4.8% (for OpenSHH, WuFTP, and NullHttpd)
- Hardware area overhead
  - 30% on FPGA (but only 7.5% of equivalent ASIC gates)
  - ASIC implementation routes not constrained to pre-placed wires (as it is on FPGA) and can be placed more efficiently

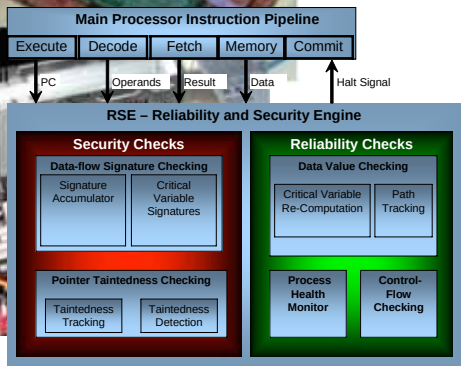
Executing on a Coprocessor: The Trusted Illiac Node Architecture



Deployment in Trusted Illiac

Initial Cluster
•256 Linux nodes

FPGA-based hardware



Trusted Illiac Node for advanced hardware development

Reliability and Security Engine
•DLX (MIPS ISA)
•Leon3 (SPARC ISA)

- Application-specific detectors
  - Reliability – process health monitor, data value checking
  - Security – dataflow signature checking, pointer-taintedness checking
- Definition of hardware-software interfaces
  - P2P Streaming application
  - Model-driven trust management
- Integration of hardware accelerators with Linux OS