Software Fault Isolation for Robust Compilation

Ana Nora Evans, University of Virginia and INRIA Paris*

---

Robust Compilation

Goals:
1. Allow reasoning about safety properties at the source level.
2. Limit the potential damage of corrupt (low-level) libraries.

A low-level compromised component cannot cause more harm than a source level one could.

Implementation:
- Proof-of-concept two-pass compiler
- Galina with Coq proofs for source to intermediate pass
- One back-end using Software Fault Isolation (presented here), another using hardware tags

Memory unsafe source language with undefined behavior, enriched with a notion of component with the following constraints:
- A component can write only in its own memory.
- Each component defines an interface
  - A list of procedures others can call
  - A list of procedures it can call
- Execution can be transferred to a component only by calls allowed by interface
- Returns from cross-component calls can cause more harm than a source level libraries.

Goals:
- Formal Definition (Source to Intermediate):
  - Reserved (Code)
  - Allow reasoning about safety
  - Proof

---

Software Fault Isolation

Compiler transformation to prevent:
1. Unsafe memory writes
2. Unsafe cross-component jumps

Cross-Component Call Stack

Memory Layout

Transformations Examples

Protection of cross-component stack:
- Writes only in the data slots of the component prevent.
- Code injected only in data slots
- Protected stack smashing
- Execution from code slots only prevents
- Execution of any possible injected code
- Alignment and the halt guard prevent RDP

Related Work:

---

Proposed trace in intermediate semantics

Compiler Proof

Predicate trace in intermediate semantics

Formal Definition (Intermediate to Target):

Intermediate Trace and Execution Result

Component Writer

Intermediate Code

Target Code

Result

---

Property Based Testing

Generators of intermediate programs:
- Used frequency constraints to increase the likelihood of tested behaviors
- Generated valid intermediate memory
- Generated groups of instructions to avoid undesirable undefined behaviors like:
  - Cross (random stack) (random register)
  - (random memory)
  - Cross (random register)
- Generated desired undefined behaviors, for
  - Paralysis of certain instructions
  - (random stack) (random register)

Compiler correctness:
- Used CompCert definition based on traces of cross-component calls and return
- All undefined behaviors allowed, thus the target program can produce larger traces
- Property that does not terminate in a maximum number of steps
- Many random with empty trace

---

Future Work

Write and test the semantics of a real RISC machine (e.g., Atmel AT91 microcontroller)