Why Information-flow is Different From – and harder than – Verifying other kinds of Properties

Steve Zdancewic
University of Pennsylvania
Where I am Coming From

• Security-Oriented Languages
  – Information-flow specification/verification
  – Authorization policies

• Low-level memory safety in legacy C code: SoftBound / CETS
  – Instrumenting LLVM code against buffer overflows

• Verified LLVM project
  – Coq model of the LLVM IR for verified program transformations / optimizations
Plan

• What do I mean by “information flow”
• Why it is different than usual “properties”
  – For a technical meaning of properties
• Ramifications?
• High-level overview of PL techniques that might apply to hardware
Information-flow Policies

- Protecting against leakage of confidential information
  - Noninterference (& many variants in the literature)

[Goguen & Meseguer; Lamport; Manna & Pnueli; etc.]
Information-Flow Policies in Hardware?

• Useful to prevent design mistakes?
  – Ensure that testing infrastructure doesn’t impact consumer-observable behavior?
  – Check properties of hardware that manipulates secrets (e.g. crypto hardware does not leak private keys)

• Useful in the context of untrusted components
  – 3rd party IP?
Program Properties

• Often, verification is concerned with:
  – Safety: something bad \textit{does not} happen
  – Liveness: something good \textit{eventually does} happen

• A property can be specified by a predicate on a \textit{single trace} of the system’s execution.
  – Model checking as a validation mechanism
  – Refinement/simulation as translation correctness

[Alpern & Schneider; Lamport; Manna & Pnueli; etc.]
Information-flow is not a Trace Property

- Information-flow security constraints are usually specified by correlating two executions of the system.
- Intuitively: Information can leaked by observing that some event didn’t happen.

```java
secret = read_secret_input();
public = 0;
if (secret > 10) {
    public = 1;
}
// public = 1 iff secret > 10
```

- Information-flow is a property about a set of possible traces.
Given the same public input and different secret inputs, does the system produce the same public outputs?
(Somewhat) More Formally

• A system $P$ is information-flow secure, if, for all attack contexts $C_1$ and $C_2$:

\[ C_1[P] \approx C_2[P] \]

• Here $\approx$ is any notion of “system equivalence”
  – It characterizes the “level of abstraction” (i.e. the observational power of the attacker/system)

• **Attack context**: tests $P$ by supplying different secrets
  – More generally: what influence does the attacker have on $P$?

• Related techniques from the PL world: relational parametricity and logical relations
  – Used to reason about modularity and abstraction in programs
Verifying Software Information Flow Policies

• Static analysis:
  – post-hoc verification of existing code
  – difficult, but possible in practice

• Type systems:
  – programming language support for creating software that is secure by construction
  – type safety implies information-flow security
  – conservative: rules out some good programs

• (Also a growing literature on dynamic enforcement mechanisms)
What about Refinement?

• Corollary: Information-flow properties are *not* preserved by refinement.

• Trivial Example:

```java
secret = read_secret_input();
public = ???; // (unspecified, i.e. nondeterministic)
```

```java
secret = read_secret_input();
public = secret; // (refinement allowed by resolving nondeterminism)
```

• Solution(?): Include information-flow properties in the specification property to be preserved by refinement.
Model Checking?

• Desirable information-flow policies are not expressible in mu calculus.
  – mu-calculus is the specification logic supported by many standard model checkers.

• One possibility:
  – Model-checking a “doubled” version of the system
    \[ \phi(P) \rightarrow \phi'(P_1 | P_2) \]
  – State-space blowup?
Challenges (at the Software Level)

• Information-flow specification is relative to some level of abstraction.
  – Detail of the model impacts the “observations” that the attacker can make. What is the choice of $\approx$?
  – Choice affects the strength of the result & difficulty of verification

• Policy is hard:
  – What information is confidential?
  – Typically, noninterference (and its relatives) are not the desired policy.
  – Need more precise specifications of permissible information flows
Hardware Introduces More Problems

- Parallelism / Concurrency
  – notoriously tricky in software
- Speculation, out-of-order execution
- Relaxed memory consistency models
- Caching / Timing Effects
- Power channels
- …

- How do these interact with information flow?
  – Even specifying the desired property might be hard.
Conclusions

• Information-flow is *not* a safety (or liveness) property
  – not directly amenable to model checking
  – must be careful with refinement

• Programming languages and security communities have made significant advances in formalizing information-flow properties at the software level.
  – Can these techniques be useful for hardware too?