Effective Blame for Information-Flow Violations

Dave King
SIIS Laboratory
University Park, PA
How can we trust programs are well-behaved?

- They don’t copy your secret data to the network or share it with just anyone

No easy solution

- Operating system controls are too coarse-grained
- Requires instrumenting programs
Formally verify that a program satisfies noninterference (Denning and Denning, 1977)

Compiler certifies that a program behaves correctly.
- Enforce information-flow security: high security information does not affect any low observable event.

Explicit information leaks: leaking $f(\text{secret})$
(for some function $f$, i.e. releasing secret $\times 2$, etc)

Implicit information leaks: leaking any other information about the secret (covert channels)

Are both kinds of leaks equal?
Let $L$ be a low security variable and $H$ be a high security variable.

Explicit Flows (data flow)
\[
L := H, \quad L := H \times 2, \quad L := f(H)
\]

Implicit Flows (control flow)
\[
\text{if } H == 0 \text{ then } L := 1
\]
\[
\text{If } (\text{password} == \text{guess}) \text{ then output } \text{“correct password!”}
\]

Visualize an information flow from sources to sinks

Sources: locations where secret data is stored in the system

Sinks: locations where data is written out of the system
Methodology: view security enforcement as a type safety property (Volpano, Smith, and Irvine 1996)

Type safety: ML, Java, …

Use advancements in types to aid security checking

Most security-typed language implementations use a constraint-based type system

Compiler generates constraints from source code

Solvable constraints = secure program
Security Types

When checking a program, view security annotations as "seed labels."

- Declaring an integer as \{Secret\} gives it the type \text{int\{Secret\}}.

- Type system prevents outputting an \text{int\{Secret\}} on a public channel.

Infer the rest of the labels -- what kind of label analysis is done?

- Different security types on different method calls? (Context sensitive)

- Different security types for different object instances assigned to the same pointer (Object sensitive)
Implementations of practical security-typed languages allow building practical systems with end-to-end security guarantees.

- Mail clients (Hicks et al. 2006)
- Servlet framework (Chong et al. 2007)
- Remote voting system (Clarkson et al. 2008)
Other Techniques

Noninterference is not the only security property that’s important for code to satisfy…

Automatic placement of reference monitor hooks (Ganapathy et al 2005)

FABLE: general language-based enforcement of complete mediation (only policy functions can rewrite labeled types) (Swamy et al 2008)

General static analysis tools for security: detecting buffer overflows, data races, etc. Splint, commercial tools (fortify)
Security-typed language implementations

- Jif (Java variant)
- Flow Caml (based on an OCaml subset)

Other tools for language-based security

- CQUAL: type qualifiers (user pointer vs kernel pointer; type error if user pointer is used where kernel pointer excepted)

Program bugs are often security bugs -- most static checkers address security vulnerabilities (buffer overflow, data races, ...).
Current Challenges

Building bigger end-to-end systems

What general techniques are there for building large secure systems?

Verifying security throughout the actual compilation process from source code to verifiable bytecode

Annotate compiled code with proof of security (similar to proof-carrying code)

Still getting this for types... (Chlipala 2007); some plans for an information flow certifying compiler (Naumann et al. 2006)
Noninterference is too strong

Release information through *declassification* statements

Can we come up with formal systems that allow some information to be leaked?

Survey paper: (Sabelfeld and Sands 2006)
One Big Challenge...

For a number of reasons, security-typed languages require their code to be written from scratch:

- Not restricted to issues faced by original language designers
- Better efficiency from intraprocedural analysis (one procedure at a time)

Can we retrofit information flow security guarantees onto existing code?

Compare: can we add reference monitor guarantees to existing code?
Most programs cannot be simply transformed into a security-typed equivalent.

Code is written without security in mind

Declassification points might be unknown

Hard for a static checker to verify noninterferent code such without a lot of hand holding

*How best to detect and view information flow errors?*  
(tod...
The Blame Problem

Need to find information-flow violations in programs so that we can fix them.

Jif and Flow Caml do not have good error reporting mechanisms.

- Long-studied history of improving error messages in ML (Wand 86, many others)

The cause of an error is often unclear.

- Information-flow errors are difficult to understand

In legacy code, errors can span many procedures
Jif generates constraints from the source code and attempts to solve them with a variant of the Rehof-Mogensen constraint solver (Rehof and Mogensen, 1996). Constraints have the form $\tau \leq \beta$ or $\tau \leq L$: $\beta$ is a label variable, $L$ is a security label, and $\tau$ is a label expression consisting of a join of labels and label variables.

RM Solver works in two phases

• first ensures constraints of the form $\tau \leq \beta$ are satisfied by raising each variable $\beta$ high enough.

Then check constraints of the form $\tau \leq L$. If any one of the constraints is not satisfied, fail.

If approach: blame the failed constraint $\tau \leq L$. 
Current Problems with Blame

Information-flow security ensures that each source with security level $L$ only flows to sinks with security level $L'$ such that $L' \geq L$ holds.

Blaming the failing constraint will only blame a secure sink.

Does not report the actual cause of an information-flow error!
We claim that an error report for an information-flow violation is ideally:

- **complete** (the error report contains everything that caused a violation)
- **minimal** (the error report does not contain any extra information)
Introduce the blame dependency graph, a data structure that records information during a run of the Rehof-Mogensen solver. When the first phase of the RM solver adjusts the variable $\beta$ because of the constraint $\tau \leq \beta$, mark dependency between the variables that occur in $\tau$ and the variable $\beta$. This stored information allows us to determine why a constraint failed.
Blame dependency graph example + constraint solving example from paper will go here.
Recursively use blame dependency graph to explain why constraint $\tau \leq L$ failed. Either $\tau \leq L$ is unsatisfiable or became unsatisfiable at some point during the run of the RM solver.

Determine $\tau' \leq \beta$, the reason why $\tau \leq L$ failed.

Next recursively determine why $\tau' \leq L$ failed.

Works because of monotonicity of join -- $\beta$ was raised too high because either $\tau' \leq L$ is unsatisfiable or some variable $\alpha \in Vars(\tau')$ was raised above $L$. 
Theoretical Results

Completeness: if $\text{Recursive Explain} (\tau \leq L) = X$, then the set $X \cup \{ \tau \leq L \}$ is unsatisfiable.

Minimality: there is no $Y \subset X$ such that $Y \cup \{ \tau \leq L \}$ is unsatisfiable.

Proof idea: sets generated by $\text{Recursive Explain}$ can be ordered as $\{ \tau_0 \leq \beta_0, \ldots, \tau_n \leq \beta_n \}$ where each $\beta_i$ is distinct and for each $i$, $\beta_i \in \text{Vars}(\tau_i)$. 
Expanded Jif compiler to support interprocedural security label analysis.

“Vanilla Jif” requires manual annotation of the security behavior of each procedure.

Determine security requirements of each procedure using summary constraints.

We found that adding more label variables to the compiler for concepts such as the program counter (used to track implicit flows).
Compared error traces with known method for finding information-flow errors involving slicing (Hammer et al 2005) based on path conditions. Found that our approach gave small error traces. Catalogued non-exceptional error traces in email server, SQL database (among others). Found that context sensitivity was required for precise analysis; unfortunately this can get expensive.
Blame procedure allows programmer to manually examine information flows.

However, this gets unreasonable for larger programs.

Idea: we can summarize information leaks using declassification statements.

Work in progress: placing a minimum set of declassifiers (serve as “escape hatches” for security type system) automatically.

Methodologies: (1) SAT solver (2) minimum-cut based graph approach.
False Positives

Academic security tools that have been used in the “real world” have focused on explicit flows of information.

ex: CQUAL for verifying hook placement and finding user-kernel bugs in the Linux kernel.

What about implicit flows?

Password check can be characterized as an implicit flow, so implicit flows can leak real information.
Using interprocedural label analysis from FSE work, we examined a number of codebases to determine the impact of implicit flows.

Findings: most implicit flow alarms were caused by exceptions that could not be thrown at runtime.

```
v = obj.foo(); throws an exception if obj is null
```

If we cannot statically prove that obj is never null, sound security analysis must assume that this program path is possible!

Basic issue with Java/code written in imperative languages -- many possible runtime paths from very little code.
Very few errors were caused by the type-based nature of the analysis.

Suggests that finer-grained analyses (object-sensitive) might not be the answer.

Most runtime exceptions due to null pointer exceptions, array index out of bounds exceptions, and class cast exceptions.

Need to deal with this false positive ratio when retrofitting code for information-flow security.