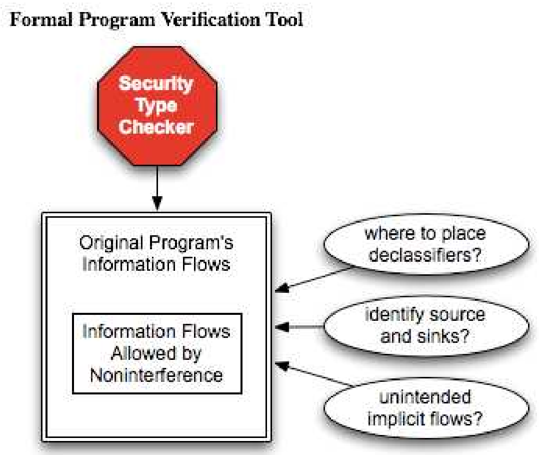
Retrofitting Programs for Information-Flow Security

Dave King, Trent Jaeger, Boniface Hicks, Patrick McDaniel (Penn State),
Michael Hicks (University of Maryland), Somesh Jha (University of Wisconsin),
Sanjit A. Seshia, Susmit Jha (UC Berkeley)

Programs are trusted to enforce system security policies on the data that they store and output. A recent development in creating *verifiably* secure programs are *security-typed languages* (STLs), where a special security checker enforces label properties during program compilation. Most commonly, these programs enforce a weakened form of noninterference, which prevents low security data from A number of programs have been built using security-typed languages, including an email server (JPMail), a remote voting system (Civitas), and a framework for servlets that maintain user security (SIF). However, most programs are not written with information-flow guarantees in mind, making it hard to verify their security properties. We propose methods for retrofitting programs to enforce information-flow security goals in a semi-automatic fashion. We focus on improving *security analyses* and *tool support* for security-typed language programmers.

Challenges

- **Determining the correspondence between security policy and program code:** before any analysis can be done, security-relevant *sources* and *sinks* need to be identified along with the policies that they require.
- **Understanding and resolving information-flow errors:** information-flow errors in existing code can be quite complex and span many procedures. A recent publication [1] investigates a method for giving *complete* and *minimal* error explanations for information-flow errors.
- **Declassification:** Real programs release information through declassification. How can we choose the best locations for escape hatches in the security analysis? We have preliminary work on automatically placing declassifiers with the aid of a SAT solver [2].
- **Unintended implicit flows:** Programs are not written with full noninterference in mind, and so enforcing information-flow security on them may reveal that programs reveal secret information in a surprising number of ways.


Formal Program Verification Tool
Security Type Checker
Original Program's Information Flows
Information Flows Allowed by Noninterference
where to place declassifiers?
identify source and sinks?
unintended implicit flows?

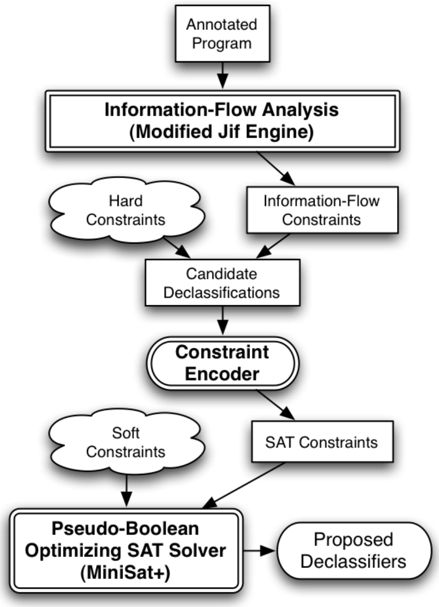
Understanding Errors

To explain information-flow errors, we add a data structure known as the *blame dependency graph* to the Rehof-Mogensen constraint solver (a worklist constraint solver that can be used to solve information-flow constraints). The blame dependency graph can be queries to find a complete and minimal error set for an information-flow violation. Through experimental observation, we have displayed that each error trace contains a fixable program statement: a program statement that uniquely cause the specific error.

$$\begin{tikzcd}
(\beta_0, 0, \bot) \arrow[d, "L_1 \le \beta_0"] & (\beta_1, 0, \bot) \arrow[d] & (\beta_2, 0, \bot) \arrow[d] \\
(\beta_0, 1, L_1) \arrow[d] \arrow[dr, "L_2 \sqcup \beta_0 \le \beta_1"] & (\beta_1, 1, \bot) \arrow[d] & (\beta_2, 1, \bot) \arrow[d] \\
(\beta_0, 2, L_1) \arrow[d] \arrow[drr, "\beta_0 \sqcup \beta_1 \le \beta_2"] & (\beta_1, 2, L_1 \sqcup L_2) \arrow[d] \arrow[dr, "\beta_0 \sqcup \beta_1 \le \beta_2"] & (\beta_2, 2, \bot) \arrow[d] \\
(\beta_0, 3, L_1) & (\beta_1, 3, L_1 \sqcup L_2) & (\beta_2, 3, L_1 \sqcup L_2)
\end{tikzcd}$$

Placing Declassifiers

Programs release information in a number of ways. In an information-flow secure language (such as Jif), declassifiers are used to allow otherwise illegal information flows. We are developing a declassifier placement system to place declassifiers in a program automatically. A set of automatically-placed declassifiers can give the programmer knowledge of what illegal information flows that a program requires to function. If these information flows do not match what is expected, our blame framework [1] can be used to narrow down the ultimate source of an unexpected declassifier.


Annotated Program
Information-Flow Analysis (Modified Jif Engine)
Hard Constraints
Information-Flow Constraints
Candidate Declassifications
Constraint Encoder
Soft Constraints
SAT Constraints
Pseudo-Boolean Optimizing SAT Solver (MiniSat+)
Proposed Declassifiers

Our declassifier placement system system is based on a SAT solver: information-flow constraints encoded, along with possible declassifiers, as a psuedo-Boolean SAT problem. We use *hard constraints* to filter out infeasible declassifier locations and *soft constraints* to rank candidate declassifiers as to which declassification locations are more preferable than others. A minimal solution to the psuedo-Boolean SAT program corresponds to a minimal set of declassifiers of minimal weight.

Future Work

Most illegal information flows will be caused by a difference between the program's existing security policy and noninterference. Off-the-shelf applications contain a large number of illegal information flows that will either have to be permitted through declassifiers or manually rewritten. We will focus on applying our results to build tools that will help ease this process.

References

[1] Dave King, Trent Jaeger, Somesh Jha, Sanjit A. Seshia. Effective Blame for Information Flow Violations. Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2008). To appear.
[2] Dave King, Susmit Jha, Trent Jaeger, Somesh Jha, Sanjit A. Seshia. On Automatic Placement of Declassifiers for Information-Flow Security. Dave King, Susmit Jha, Trent Jaeger, Somesh Jha, and and Sanjit A. Seshia. Technical Report NAS-TR-0083-2007, Network and Security Research Center.

Sponsored by AFRL/IARPA