Information Flow Security in Imperative Languages

Robert Grabowski

Institut für Informatik
Ludwig-Maximilians-Universität München

14. Kolloquium Programmiersprachen und Grundlagen der Programmierung
Timmendorfer Strand, Oktober 2007

Gefördert durch DFG-Projekt InfoZert (Be 3712/2-1)
What are secure information flows?

- control flow of information between objects of a computing system (like variables, files, channels)
- restricting access to objects is not flexible enough
What are secure information flows?

- assign confidentiality domains to objects
- specify secure information flows using a *flow policy*
  - e.g. allow flows from public to private domain, but not vice versa
Secure flows in imperative languages

A simple WHILE language

\[ x \in X \]
\[ v \in V = \mathbb{N} \]
\[ e ::= x \mid v \mid e_1 + e_2 \mid \ldots \]
\[ C ::= x := e \mid C_1 ; C_2 \mid \text{skip} \mid \text{if } e \text{ then } C_1 \text{ else } C_2 \mid \text{while } e \text{ do } C \]

program states: \( \sigma, \tau : X \rightarrow \mathcal{V} \)
given semantics: \( \llbracket e \rrbracket_\sigma \) and \( \sigma \xrightarrow{C} \tau \)
Secure flows in imperative languages

Identifying secure flows

- security objects: variables
- classified into low and high domain: $\mathcal{X}_{low} \cup \mathcal{X}_{high} = \mathcal{X}$
- flow policy: low $\rightsquigarrow$ high, but high $\nrightarrow$ low

Data flow examples

assume $l \in \mathcal{X}_{low}, h \in \mathcal{X}_{high}$

\[
\begin{align*}
    h &:= l \quad \text{(ok)} \\
    l &:= h \quad \text{(insecure)} \\
    l &:= h \mod 2 \quad \text{(insecure)} \\
    \text{if } h > 0 \text{ then } l &:= 1 \text{ else } l := 3 \quad \text{(insecure)}
\end{align*}
\]
Non-interference

- initial values of *high* variables may not affect final values of *low* variables
- equivalently: when observing only *low* variables, the behaviour of the program must be deterministic

\[
\begin{array}{c}
\begin{array}{c}
\sigma \\
\hline \\
\sigma' \\
\hline
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
C \\
\hline
C
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
\tau \\
\hline \\
\tau'
\end{array}
\end{array}
\]
Non-interference

Definition
Two states $\sigma, \sigma'$ are indistinguishable, written $\sigma \sim \sigma'$, if for all $x \in X_{low}$: $\sigma(x) = \sigma'(x)$.

A program $C$ is secure if whenever $\sigma \sim \sigma'$ and $\sigma \xrightarrow{C} \tau$ and $\sigma' \xrightarrow{C} \tau'$, then $\tau \sim \tau'$. 
Other types of information flows

flows not covered by non-interference definition:

- termination behaviour
- timing aspects
- resource usage
- ...

here: only flows between variables (direct and implicit)
Static flow analysis

- type system by Volpano, Smith, Irvine (1996)
- check assignments for invalid direct flows
- implicit flows: do not allow assignment to low variables under high guard
  \[
  \text{if } h > 0 \text{ then } l := 1 \text{ else } l := 3
  \]
Typing expressions

\[ k ::= \text{low} \mid \text{high} \]

\[ \vdash e : k \]

\[ \frac{\text{Vars}(e) \cap \mathcal{X}_{\text{high}} = \emptyset}{\vdash e : \text{high}} \]

\[ \frac{\vdash e : \text{high}}{\vdash e : \text{low}} \]

- if \( \vdash e : \text{low} \), then \( e \) does not depend on \( \text{high} \) variables
Typing programs

\[ k ::= \text{low} \mid \text{high} \quad h \in X_{\text{high}}, \ l \in X_{\text{low}} \]

\[
[k] \vdash C \]

\[
[\text{high}] \vdash h := e \\
[\text{low}] \vdash l := e
\]

\[\vdash e : \text{low} \]

Soundness: if \([\text{low}] \vdash C\), then \(C\) is secure, i.e. non-interferent
Typing programs

\[ k ::= \text{low} \mid \text{high} \quad h \in \mathcal{X}_{\text{high}}, \ l \in \mathcal{X}_{\text{low}} \]

\[ [k] \vdash C \]

\[ [\text{high}] \vdash h := e \quad \vdash e : \text{low} \quad [\text{low}] \vdash l := e \]

- \( k \) is a lower bound of the types of variables assigned in \( C \)
Typing programs

\[ k ::= \text{low} \mid \text{high} \quad h \in \mathcal{X}_{\text{high}}, \; l \in \mathcal{X}_{\text{low}} \]

\[ [k] \vdash C \]

\[ [\text{high}] \vdash h ::= e \quad [\text{low}] \vdash l ::= e \]

\[ \vdash e : \text{low} \]

\[ \vdash e : k \quad [k] \vdash C_1 \\ [k] \vdash C_2 \]

\[ [k] \vdash \text{if } e \text{ then } C_1 \text{ else } C_2 \]

\[ \vdash e : k \quad [k] \vdash C \]

\[ [k] \vdash \text{while } e \text{ do } C \]

- \( k \) is a lower bound of the types of variables assigned in \( C \).
Typing programs

\[ k ::= \text{low} \mid \text{high} \quad h \in \mathcal{X}_{\text{high}}, \ l \in \mathcal{X}_{\text{low}} \]

\[ [k] \vdash C \]

\[ [\text{high}] \vdash h := e \quad [\text{low}] \vdash l := e \]

\[ \vdash e : \text{low} \]

\[ \vdash e : k \quad [k] \vdash C_1 \quad [k] \vdash C_2 \quad \vdash e : k \quad [k] \vdash C \]

\[ [k] \vdash \text{if } e \text{ then } C_1 \text{ else } C_2 \]

\[ [k] \vdash \text{while } e \text{ do } C \]

\[ [\text{high}] \vdash \text{skip} \]

\[ [\text{low}] \vdash C \]

\[ [k] \vdash C_1 \quad [k] \vdash C_2 \quad [\text{high}] \vdash C \]

\[ [\text{low}] \vdash C \]

\[ \square k \text{ is a lower bound of the types of variables assigned in } C \]
Typing programs

\[ k ::= \text{low} \mid \text{high} \quad h \in \mathcal{X}_{\text{high}}, \ l \in \mathcal{X}_{\text{low}} \]

\[ [k] \vdash C \]

\[ [\text{high}] \vdash h := e \quad [\text{low}] \vdash l := e \]

\[ [k] \vdash e : \text{low} \]

\[ [k] \vdash C_1 \quad [k] \vdash C_2 \]

\[ [k] \vdash \text{if } e \text{ then } C_1 \text{ else } C_2 \]

\[ [k] \vdash \text{while } e \text{ do } C \]

\[ [\text{high}] \vdash \text{skip} \quad [\text{low}] \vdash C \]

\[ [high] \vdash C_1 \quad [k] \vdash C_2 \]

\[ [high] \vdash C \]

\[ [k] \vdash C_1 \ ; \ C_2 \]

\[ k \text{ is a lower bound of the types of variables assigned in } C \]

\[ \text{Soundness: if } [\text{low}] \vdash C, \text{ then } C \text{ is secure, i.e. non-interferent} \]
Further extensions

Richer language

- recursive functions
- references, objects, inheritance
- exceptions

Improve analysis

- more complete type system
- program logics
Our goal: secure flow framework

secure software design

- extend UML with specifications for security domains and flow policies
- translate requirements to language level

proof-carrying code architecture

- static flow analysis during compilation
- distribute proof as certificate for compiled code
- client (e.g. mobile device) can easily check whether certificate is correct and satisfies system policy
Limitations of static analysis

Problem: data objects only known at run-time

- example: program with file I/O
- to ensure non-interference for file contents, need to know which files are manipulated
- requires model of file names
- but: file names are interpreted strings
Dynamic labels

Solution

- associate data objects with labels (type variables)
- actual type available at runtime, may be verified against policy
- prove that program is secure for every type instantiation

Extended syntax

\[ C ::= \ldots \mid \text{with } (x_\alpha \mapsto f(e)) \text{ do } C \mid \text{if } \alpha \rightsquigarrow \beta \text{ then } C_1 \text{ else } C_2 \]

Example

\[
\text{with } (f_\alpha \mapsto \text{openFile}(fn)) \text{ do } \\
\quad \text{if } \alpha \rightsquigarrow \text{low} \text{ then } l := f \text{ else skip}
\]
Other practical issues

- non-interference is often too strict
  - certain situations require a controlled way to declassify and release sensitive information
  - need intuitive formalisation
- concurrent and distributed systems
- portability: programmer’s intended policy vs client policy

some issues covered by Jif (Java with information flows)
Summary

- restrict flow of information, but not access to it
- non-interference formalises confidential data confinement
- static program analysis using type system
- embed in framework for flow-secure software engineering
- challenge: adopt security notion to real-world software
- contribution: dynamic labels for run-time objects