Secure Information Flow as a Safety Property

Gérard Boudol

INRIA Sophia Antipolis

Software Security

To prevent application software from running into security violations:

- defensive attitude: protection of confidential information and precious resources.

↪ analysis of binary code, run-time checks.

Severe limitations on possible interactions.

- constructive attitude: build and use software offering security guarantees, that can be trusted.

↪ provide tools to design, develop and maintain secure software.

Aim: security-minded programming primitives and (static) analysis techniques of programs to build “safe-by-construction” software.

Focus: Confidentiality

(Integrity is dual.)

Information "containers" – files, database entries, library functions, memory locations... – are classified into (ordered) security levels, e.g.

$$institution \prec group(s) \prec user(s) \prec root$$

with

- access control: a program should only read information it has the right to access.
- information flow control: a program should not disclose secret information.

Flow policy: $\ell \preceq \ell'$ says information is allowed to flow from level ℓ to level ℓ'.

PROGRAMMING SECURITY (1/2)

Some security-minded programming constructs, to manage access control:

- (**enable** ℓ **in** P) grants the (read) access right ℓ to P.
- (**restrict** P **to** ℓ) dual, restricts the access right of P by ℓ.
- (**test** ℓ **then** P **else** Q) tests whether access right ℓ is granted or not, behaves accordingly as P or Q.

cf. JAVA "stack inspection."

Programming Security (2/2)

and to manage information flow:

- $[\ell_0 \searrow \ell_1]P$ tests whether the confidentiality level of P is less than ℓ_0, if yes turn it into ℓ_1 – declassification.
- $(\text{flow } F \text{ in } P)$ enrich the current flow policy by F for running P.
- $(\text{revoke } F \text{ in } P)$ dual, executes P without the flow policy F.
- $(\text{check } F \text{ then } P \text{ else } Q)$ tests whether the flow policy F is granted, and branches.

e.g. JIF has **declassify**$(M, \ell) = [\top, \ell \searrow M]$.

Example (Declassification)

The governmental software for computing and collecting taxes (on the salaries and revenues), while manipulating private data, should be allowed to publish statistical informations, like

$$average\ tax\ amount = [gvt \searrow public]\ \frac{\sum_{individuals} tax\ amount_i}{nb\ individuals}$$

with

$$public \prec individual(s) \prec gvt$$

Standard Semantics

for secure information flow: non-interference – "variety in secret input should not be conveyed to public output".

- operational semantics: $(P, \mu) \Downarrow \nu$. Starting from memory μ, program P terminates with memory ν.
- memory: mapping program variables, with security levels, to values.
- low equality of memories:

$$\mu =^{\preceq \ell} \nu \quad \Leftrightarrow_{\text{def}} \quad \forall x. \forall \ell'. \; \ell' \preceq \ell \;\Rightarrow\; \mu(x_{\ell'}) = \nu(x_{\ell'})$$

- non-interference: P is secure from the information flow point of view iff for any security level ℓ

$$\mu =^{\preceq \ell} \nu \;\&\; (P, \mu) \Downarrow \mu' \;\&\; (P, \nu) \Downarrow \nu' \;\Rightarrow\; \mu' =^{\preceq \ell} \nu'$$

PROBLEM

The non-interference property is inadequate:

- incompatible with declassification, inappropriate for revocation.
- does not formalize the intuitive notion of secure information flow, which is

  "*one should not put in a public location a value elaborated using confidential information,*"

  a safety property – "nothing bad will happen."

Standard static analysis techniques (security type systems) guarantee a stronger property than non-interference: no "programming error", unlike

$$P \text{ ; } x_{public} := y_{secret} \text{ ; } Q$$
$$x_{public} := (\text{if } y_{secret} \text{ then } P \text{ else } Q)$$

Towards Secure Information Flow

as a safety property: define a monitored operational semantics (*cf.* Fenton's Data-Mark-Machine 1974) where

- one maintains the current reading clearance (*cf.* "stack inspection") and the current flow policy;
- one keeps track of the level of knowledge acquired while computing, i.e. the current confidentiality level;
- one checks that
  - when reading in the memory, the current reading clearance is enough;
  - when writing in the memory, there is no illegal flow, i.e. the level of acquired knowledge is less than the level to update.

Objectives

Given a "security programming language:"

- define the monitored semantics;
- ↪ a notion of run-time security violation: being stuck on a security check.

A program is **secure** when it successfully passes, for any execution, all the security checks (Fenton 74).

- define static analysis methods – a security type and effect system;
- prove type safety: no run-time error for typable programs.
- ↪ run-time monitoring is not needed for typable programs.
- show that, without security programming constructs, the safety property implies non-interference.

A Language

à la ML: functional and imperative (∼ JAVA: methods and mutable fields) with programming constructs for security:

$$
\begin{array}{llll}
V,\ W\ldots & ::= & x \mid u_\ell \mid \lambda x M \mid tt \mid ff \mid () & \textit{values} \\
M,\ N\ldots & ::= & V \mid (\textsf{if } M \textsf{ then } N \textsf{ else } N') \mid (M N) & \textit{expressions} \\
 & \mid & M\,;N \mid (\textsf{ref}_\ell\, N) \mid (!\, N) \mid (M := N) & \\
 & \mid & (\textsf{restrict } M \textsf{ to } \ell) \mid (\textsf{enable } \ell \textsf{ in } M) & \\
 & \mid & (\textsf{test } \ell \textsf{ then } M \textsf{ else } N) & \\
 & \mid & (\textsf{flow } F \textsf{ in } M) \mid (\textsf{revoke } F \textsf{ in } M) &
\end{array}
$$

where ℓ is a security level, and F a flow policy.

Note: the construct $[\ell_0 \searrow \ell_1]M$ is derivable from $(\textsf{flow } F \textsf{ in } M)$. We omit $(\textsf{check } F \textsf{ then } M \textsf{ else } N)$.

Security Policies

- security levels ℓ are sets of principals.
  Reverse inclusion ordering: hierarchy for access rights and information flow.
  $\ell \supseteq \ell'$ means ℓ' is more restrictive than ℓ.
- flow policies F are binary relations on principals: if $p\,F\,q$ information accessible by principal p may flow, according to policy F, to principal q.

➥ A lattice structure:

$$\ell \preceq_F \ell' \quad \Leftrightarrow_{\mathrm{def}} \quad \forall q \in \ell'\, \exists p \in \ell.\ p\, F^*\, q$$

with join $\ell \curlyvee_F \ell'$ and meet $\ell \curlywedge_F \ell'$.

Monitored Semantics (1/4)

In the context of a reading clearance **rc** and a flow policy F, and starting with a knowledge level **pc** (initially $\bot$) and a memory μ, the expression M reduces to a value V, having acquired knowledge level ℓ, and updates the memory into ν:

$$\mathsf{rc}; F \vdash (\mathsf{pc}, M, \mu) \Downarrow^{\mathsf{m}} (\ell, V, \nu)$$

Some cases:

$$\frac{\mathsf{rc}; F \vdash (\mathsf{pc}, M, \mu) \Downarrow^{\mathsf{m}} (\ell', tt, \mu') \qquad \mathsf{rc}; F \vdash (\ell', N_0, \mu') \Downarrow^{\mathsf{m}} (\ell, V, \nu)}{\mathsf{rc}; F \vdash (\mathsf{pc}, (\mathsf{if}\ M\ \mathsf{then}\ N_0\ \mathsf{else}\ N_1), \mu) \Downarrow^{\mathsf{m}} (\ell, V, \nu)}$$

MONITORED SEMANTICS (2/4)

$$\frac{\begin{array}{l}\mathsf{rc}; F \vdash (\mathsf{pc}, M, \mu) \Downarrow^{\mathsf{m}} (\ell', \lambda x M', \mu') \\ \mathsf{rc}; F \vdash (\mathsf{pc}, N, \mu') \Downarrow^{\mathsf{m}} (\ell'', V', \nu') \\ \mathsf{rc}; F \vdash (\ell' \curlyvee_G \ell'', \{x \mapsto V'\}M', \nu') \Downarrow^{\mathsf{m}} (\ell, V, \nu)\end{array}}{\mathsf{rc}; F \vdash (\mathsf{pc}, (M N), \mu) \Downarrow^{\mathsf{m}} (\ell, V, \nu)}$$

$$\frac{\begin{array}{l}\mathsf{rc}; F \vdash (\mathsf{pc}, M, \mu) \Downarrow^{\mathsf{m}} (\ell', V', \mu') \\ \mathsf{rc}; F \vdash (\mathsf{pc}, N, \mu') \Downarrow^{\mathsf{m}} (\ell, V, \nu)\end{array}}{\mathsf{rc}; F \vdash (\mathsf{pc}, M \mathbin{;} N, \mu) \Downarrow^{\mathsf{m}} (\ell, V, \nu)}$$

differs from $(\lambda x N M)$.

Monitored Semantics (3/4)

$$\frac{\mathsf{rc}; F \vdash (\mathsf{pc}, N, \mu) \Downarrow^{\mathsf{m}} (\ell', u_\ell, \nu) \quad \nu(u_\ell) = V}{\mathsf{rc}; F \vdash (\mathsf{pc}, (!\, N), \mu) \Downarrow^{\mathsf{m}} (\ell \curlyvee_F \ell', V, \nu)} \; \ell \preceq \mathsf{rc}$$

$$\frac{\begin{array}{c}\mathsf{rc}; F \vdash (\mathsf{pc}, M, \mu) \Downarrow^{\mathsf{m}} (\ell_0, u_\ell, \mu') \\ \mathsf{rc}; F \vdash (\mathsf{pc}, N, \mu') \Downarrow^{\mathsf{m}} (\ell_1, V, \nu)\end{array}}{\mathsf{rc}; F \vdash (\mathsf{pc}, (M := N), \mu) \Downarrow^{\mathsf{m}} (\mathsf{pc}, (), \nu[u_\ell := V])} \; \ell_0 \curlyvee_F \ell_1 \preceq_F \ell$$

Monitored Semantics (4/4)

$$\frac{\mathsf{rc} \curlyvee r; F \vdash (\mathsf{pc}, M, \mu) \Downarrow^{\mathsf{m}} (\ell, V, \nu)}{\mathsf{rc}; F \vdash (\mathsf{pc}, (\mathsf{enable}\ r\ \mathsf{in}\ M), \mu) \Downarrow^{\mathsf{m}} (\ell, V, \nu)}$$

$$\frac{\mathsf{rc}; F \cup F' \vdash (\mathsf{pc}, M, \mu) \Downarrow^{\mathsf{m}} (\ell, V, \nu)}{\mathsf{rc}; F \vdash (\mathsf{pc}, (\mathsf{flow}\ F'\ \mathsf{in}\ M), \mu) \Downarrow^{\mathsf{m}} (\mathsf{pc} \curlyvee_F (\ell{\downarrow}_{F \cup F'}), V, \nu)}$$

where

$$\ell{\downarrow}_F = \{\, q \mid \exists p \in \ell.\ p\, F^*\, q \,\}$$

Secure Programs

- uncontrolled semantics: $\mathsf{rc}; F \vdash (\mathsf{pc}, M, \mu) \Downarrow (\ell, V, \nu)$ the same semantics without security check (F, pc and ℓ are useless). More permissive:

$$\mathsf{rc}; F \vdash (\mathsf{pc}, M, \mu) \Downarrow^{\mathsf{m}} (\ell, V, \nu) \;\Rightarrow\; \mathsf{rc}; F \vdash (\mathsf{pc}, M, \mu) \Downarrow (\ell, V, \nu)$$

- M is **secure** w.r.t. rc, F and a class $\mathcal{M}$ of memories iff

$$\mathsf{rc}; F \vdash (\bot, M, \mu) \Downarrow (\ell, V, \nu) \;\Rightarrow\; \mathsf{rc}; F \vdash (\bot, M, \mu) \Downarrow^{\mathsf{m}} (\ell, V, \nu)$$

for any $\mu \in \mathcal{M}$.

Examples

- (enable ℓ in M) **rc**-secure iff M **rc** $\curlyvee$ ℓ-secure.
- (restrict M to ℓ) **rc**-secure iff M **rc** $\curlywedge$ ℓ-secure.
- (flow F' in M) F-secure iff M $F \cup F'$-secure.
- (revoke F' in M) F-secure iff M $F^* - F'$-secure.

Types and Effects

Typing judgments:

$$\mathsf{rc}; F; \Gamma \vdash M : e, \tau$$

where

- Γ is a typing context: variables $\mapsto$ types;
- e is a security effect (r, w), where
  - r is the reading level, an upper bound of the level of significant reads M performs;
  - w is the writing level, a lower bound of the level of updates M performs;
- τ is a type:

$$\tau, ; \sigma,\ \theta \ldots \ ::= \ t \ \mid \ \mathsf{bool} \ \mid \ \mathsf{unit} \ \mid \ \theta\, \mathsf{ref}_{\ell} \ \mid \ (\tau \xrightarrow[\ell, F]{e} \sigma)$$

Some Typing Rules (1/3)

$$\frac{\mathsf{rc}; F; \Gamma \vdash M : (r, w), \mathsf{bool} \qquad \mathsf{rc}; F; \Gamma \vdash N_i : (r_i, w_i), \tau \qquad r \preceq_F w_0 \curlywedge w_1}{\mathsf{rc}; F; \Gamma \vdash (\mathsf{if}\ M\ \mathsf{then}\ N_0\ \mathsf{else}\ N_1) : (r', w'), \tau}$$

where $r' = r \curlyvee_F r_0 \curlyvee_F r_1$ and $w' = w \curlywedge w_0 \curlywedge w_1$.

$$\frac{\mathsf{rc}; F; \Gamma \vdash M : (r, w), \tau \xrightarrow[r_2, F']{(r_1, w_1)} \sigma \qquad r_2 \preceq \mathsf{rc} \qquad F' \subseteq F^* \qquad \mathsf{rc}; F; \Gamma \vdash N : (r_0, w_0), \tau \qquad r \curlyvee_F r_0 \preceq_F w_1}{\mathsf{rc}; F; \Gamma \vdash (M N) : (r', w'), \sigma}$$

Some Typing Rules (2/3)

$$\frac{\mathsf{rc}; F; \Gamma \vdash N : (r, w), \theta\ \mathsf{ref}_\ell \quad \ell \preceq \mathsf{rc}}{\mathsf{rc}; F; \Gamma \vdash (!\, N) : (r \curlyvee_F \ell, w), \theta}$$

$$\frac{\begin{array}{ll}\mathsf{rc}; F; \Gamma \vdash M : (r_0, w_0), \theta\ \mathsf{ref}_\ell & \\ \mathsf{rc}; F; \Gamma \vdash N : (r_1, w_1), \theta & r_0 \curlyvee_F r_1 \preceq_F \ell\end{array}}{\mathsf{rc}; F; \Gamma \vdash (M := N) : (\bot, w_0 \curlywedge w_1 \curlywedge \ell), \mathsf{unit}}$$

Some Typing Rules (3/3)

$$\frac{\mathsf{rc} \curlyvee r; F; \Gamma \vdash M : e, \tau}{\mathsf{rc}; F; \Gamma \vdash (\mathsf{enable}\ r\ \mathsf{in}\ M) : e, \tau}$$

$$\frac{\mathsf{rc}; F \cup F'; \Gamma \vdash M : (r, w), \tau \qquad r \preceq_{F \cup F'} r'}{\mathsf{rc}; F; \Gamma \vdash (\mathsf{flow}\ F'\ \mathsf{in}\ M) : (r', w), \tau}$$

Results

- **Type Safety:** if M is typable in the context of **rc** and F then M is secure w.r.t. **rc**, F, and the class of typable memories.

↪ no run-time security checks for typable configurations.

- for the "usual" language, without the security-minded programming constructs, if M is secure then M satifies the non-interference property.

↪ a proof that typability implies non-interference for simple programs.

Does not hold for programs with declassification. The implication is strict:

$$x_{public} := (\textsf{if } !\, y_{secret} \textsf{ then } M \textsf{ else } N)$$

where M and N always reduce to the same value.