Tractable Enforcement of Declassification Policies

Gilles Barthe[1] Salvador Cavadini Tamara Rezk[2]

[1] IMDEA Software, Madrid, Spain

[2] INRIA Sophia-Antipolis Méditerranée, France

Motivations

- Information flow policies guarantee end-to-end security
- Baseline policies (i.e. non-interference) can be enforced efficiently via type systems
- However baseline policies are too restrictive in practice
- Declassification policies allow intentional information release
  - what
  - where
  - who
- Existing enforcement mechanisms for source languages

Information flow for JVM

- Previous work proposes a lightweight information flow verifier for sequential JVM (inc. objects, methods, and exceptions)
- Transfer rules of the form (simplified)

$$\frac{P[i] = ins \quad \text{constraints}}{S, se, i \vdash st \Rightarrow st'}$$

- Assumptions on control dependence regions
- Proof follows from unwinding lemmas and inductive argument on pairs of traces
- Machine-checked implementation and verification in Coq
- Type-preserving compilation
- Extension to concurrency (for restricted fragment)

Goal

Information flow policy that:

- supports controlled release of information,
- that can be enforced efficiently,
- with a *modular proof of soundness*,
- instantiable to bytecode (here: for restricted fragment)
- can reuse machine-checked proofs (left for future work)

Policy setting

- Setting is heavily influenced by non-disclosure, but allows declassification of a variable rather than of a principal.
- Policy is local to each program point:
  - modeled as an indexed family $(\sim_{\Gamma[i]})_{i \in \mathcal{P}}$ of relations on states
  - each $\sim_{\Gamma[i]}$ is symmetric and transitive
  - monotonicity of equivalence

$$\Gamma[i] \leq \Gamma[j] \wedge s \sim_{\Gamma[i]} t \Rightarrow s \sim_{\Gamma[j]} t$$

(properties hold when relations are induced by the security level of variables)

Delimited non-disclosure

P satisfies delimited non-disclosure (DND) iff $\mathsf{entry}\ \mathcal{R}\ \mathsf{entry}$, where $\mathcal{R} \subseteq \mathcal{P} \times \mathcal{P}$ satisfies for every $i, j \in \mathcal{P}$:

- if $i\ \mathcal{R}\ j$ then $j\ \mathcal{R}\ j$;
- if $i\ \mathcal{R}\ j$ then for all s_i, t_j and $s'_{i'}$ s.t.

$$s_i \rightsquigarrow s'_{i'} \wedge s_i \sim_{\Gamma[i]} t_j \wedge \mathrm{safe}(t_j)$$

there exists $t'_{j'}$ such that:

$$t_j \rightsquigarrow^{\star} t'_{j'} \wedge s'_{i'} \sim_{\Gamma[\mathsf{entry}]} t'_{j'} \wedge i'\ \mathcal{R}\ j'$$

Local policies vs. declassify statements

One could use a construction `declassify` (e) `in` $\{\ c\ \}$ and compute local policies from program syntax:

$$[l_1 := 0]^1 \ ;\ \texttt{declassify}\ (h)\ \texttt{in}\ \{\ [l_2 := h]^2\ \}\ ;\ [l_3 := l_2]^3$$

yields

$$\begin{array}{l}
\Gamma[1](l_1) = \Gamma[1](l_2) = \Gamma[1](l_3) = L \\
\Gamma[1](h) = H \\
\Gamma[2](l_1) = \Gamma[2](l_2) = \Gamma[2](l_3) = L \\
\Gamma[2](h) = L \\
\Gamma[3] = \Gamma[1]
\end{array}$$

Where is what?

Declassification of expressions through fresh local variables:

$$\texttt{declassify}\ (h > 0)\ \texttt{in}\ \{\ [\texttt{if}\ (\ h > 0\ )\ \texttt{then}\ \{\ [l := 0]^2\ \}]^1\ \}$$

becomes

$$[h' := h > 0]^1\ ;$$
$$\texttt{declassify}\ (h')\ \texttt{in}\ \{\ [\texttt{if}\ (\ h'\ )\ \texttt{then}\ \{\ [l := 0]^3\ \}]^2\ \}$$

DND type system

- Given a NI type system $\Gamma, S, se \vdash i$; think as a shorthand for

$$\exists s_j.\ \Gamma[i], S, se \vdash S(i) \Rightarrow s_j \wedge s_j \leq S(j)$$

- Define a DND type system $(\Gamma[j])_{j \in \mathcal{P}}, S, se \vdash i$ as

$$\Gamma[i], S, se \vdash i$$

  (Note: not so easy for source languages)
- Program P is typable w.r.t. policy $(\Gamma[j])_{j \in \mathcal{P}}$ and type S iff for all i

$$\Gamma[i], S, se \vdash i$$

Soundness

If $(\Gamma[j])_{j \in \mathcal{P}}, S, se \vdash P$ then P satisfies DND.

- Policies must respect no creep up, ie $\Gamma[i](x) \leq \Gamma[\mathsf{entry}](x)$

Unwinding+Progress

- Unwinding: if $\Gamma, S \vdash_{NI} i$ then

$$(s_i \sim_\Gamma t_i \wedge s_i \rightsquigarrow s'_{i'} \wedge t_i \rightsquigarrow t'_{j'}) \Rightarrow s'_{i'} \sim_\Gamma t'_{j'}$$

- Progress: if i is not an exit point and $\mathrm{safe}(s_i)$ then there exists t s.t. $s_i \rightsquigarrow t$

$$\left.\begin{array}{c} (\Gamma[i])_{i\in\mathcal{P}}, S \vdash_{DND} P \\ s_i \sim_{\Gamma[i]} t_i \\ s_i \rightsquigarrow s'_{i'} \\ \mathrm{safe}(t_i) \end{array}\right\} \Rightarrow \exists t'_{j'}.\ t_i \rightsquigarrow t'_{j'} \wedge s'_{i'} \sim_{\Gamma[\mathsf{entry}]} t'_{j'}$$

Instantiation to JVM fragment

- Syntax:

$$
\begin{array}{lcll}
instr & ::= & \mathsf{prim}\ op & \text{primitive operation} \\
 & | & \mathsf{push}\ v & \text{push value on top of stack} \\
 & | & \mathsf{load}\ x & \text{load value of } x \text{ on stack} \\
 & | & \mathsf{store}\ x & \text{store top of stack in } x \\
 & | & \mathsf{ifeq}\ j & \text{conditional jump} \\
 & | & \mathsf{goto}\ j & \text{unconditional jump} \\
 & | & \mathsf{return} & \text{return}
\end{array}
$$

- Type system: transfer rules of the form

$$
\frac{P[i] = ins \qquad constraints}{i \vdash st \Rightarrow st'} \qquad \frac{P[i] = ins \qquad constraints}{i \vdash st \Rightarrow}
$$

where $st, st' \in \mathcal{S}^\star$

Type system for JVM fragment

- Rules:

$$\frac{P[i] = \mathsf{push}\ n}{i \vdash_{DND} st \Rightarrow se(i) :: st} \qquad \frac{P[i] = \mathsf{binop}\ op}{i \vdash_{DND} k_1 :: k_2 :: st \Rightarrow (k_1 \sqcup k_2) :: s}$$

$$\frac{P[i] = \mathsf{store}\ x \qquad se(i) \sqcup k \leq \Gamma_i(x)}{i \vdash_{DND} k :: st \Rightarrow st} \qquad \frac{P[i] = \mathsf{load}\ x}{i \vdash_{DND} st \Rightarrow (\Gamma_i(x) \sqcup se(i)}$$

$$\frac{P[i] = \mathsf{goto}\ j}{i \vdash_{DND} st \Rightarrow st} \qquad \frac{P[i] = \mathsf{return} \qquad se(i) = L}{i \vdash_{DND} k :: st \Rightarrow \epsilon}$$

$$\frac{P[i] = \mathsf{ifeq}\ j \qquad \mathsf{loop}(i) \Rightarrow k = L \vee \mathsf{term}(i) \qquad \forall j' \in \mathsf{region}(i),\ k \leq se(j')}{i \vdash_{DND} k :: \epsilon \Rightarrow \epsilon}$$

- Soundness uses unwinding and CDR properties (the latter can be checked automatically)

Laundering attacks

$$[h := h']^1 \ ; \ \texttt{declassify}\ (h)\ \texttt{in}\ \{\ [l := h]^2\ \}$$

- Such programs are insecure w.r.t. policies such as localized delimited release.
- It is possible to define a simple effect system that prevents laundering attacks:
  - judgments are of the form $\vdash_{LA} c : U, V$
  - U is the set of assigned variables
  - V is the set of declassified variables

Conclusion

- Modular method for enforcing information flow policies that support controlled information release
- Applicable to bytecode languages
- Type-preserving compilation for language with declassify statements
- Future work
    - Formal comparison with other policies
    - Multi-threaded JVM
    - Machine-checked proofs in Coq