Expressive declassification policies and modular static enforcement

David A. Naumann  
Stevens Institute of Technology

with Anindya Banerjee and Stan Rosenberg

Supported by NSF CCR-0208984, CCF-0429894
Example policy: medical record release

♦ The patient’s diagnosis is released to insurance rep, but not the doctor’s notes (both are normally secret).

♦ Preceding release, an audit log entry is made, including the patient ID and record version, as well as the IDs of the bookkeeper and insurance rep.

♦ At the time of release, both clerk and representative should be users with valid credentials to act in their respective roles.
class PatientRecord {
    int id, vsn; String{H} diagnosis; String{H} notes; }
class InsRecord { int id; String diagnosis; }

... Object release (DB db, int patID, Clerk c, InsRep r)

    precond: sys.auth(c,"clerk") && sys.auth(r,"rep")
{
    InsRecord ir := new InsRecord();
    PatientRecord pr := db.lookup(patID);
    if (pr != null) {
        log.append(c.id, r.id, pr.id, pr.vsn, "release");
        ir.id := pr.id;  ir.diagnosis := pr.diagnosis;
        return ir;
    } else { return new Msg("not available"); } }
Conformant implementation?

An automated verifier should accept the preceding code, and reject wrong versions.

Assuming:

♦ \texttt{sys.auth(...) \text{represents actual state of authentication}}

♦ \text{authentication protocol is correct}

♦ \text{integrity of audit log}

♦ \text{platform uncorrupted (program means what it says)}

♦ \texttt{release \text{executes atomically}}
Conformant implementation?

An automated verifier should accept the preceding code, and reject wrong versions.

Assuming:

♦ \( \text{sys.auth(...)} \) represents actual state of authentication

♦ authentication protocol is correct

♦ integrity of audit log

♦ platform uncorrupted (program means what it says)

♦ \text{release} executes atomically

But what does the policy even mean?
Outline of talk

♦ baseline policy and attacker model
♦ noninterference and delimited release —what
♦ gradual release —where in code
♦ our notion: conditional gradual release
♦ specification and enforcement
♦ results and future work
Baseline policy and attacker model

♦ baseline: fixed labels (e.g., $H,L$) on external variables (e.g., `System.in` and `System.out`)

♦ access control assumption: attacker cannot read or write $H$ variables

♦ attacker knows code, but it is “verified” in some sense to be defined later (e.g., attacker may provide plug-ins but these are typechecked, maybe have “proof certificates”)

[IBM 6 Sept 07 / 6]
Baseline policy and attacker model

* baseline: fixed labels (e.g., H,L) on external variables (e.g., `System.in` and `System.out`)

* access control assumption: attacker cannot read or write H variables

* attacker knows code, but it is “verified” in some sense to be defined later (e.g., attacker may provide plug-ins but these are typechecked, maybe have “proof certificates”)

* intermediate states are visible

* i.e., termination and low writes visible; not real time
Semantics of baseline policy I

Initial values of H variables do not influence final values of L variables.

Let \( \overline{l} \) be the low variables and let \( s, t \) be states. Write \( s \sim t \) iff \( s \) and \( t \) agree on the low variables. Write \( s \rightarrow^* t \) to say the program yields \( t \) from \( s \).

Non-interference: if \( s_i \rightarrow^* t_i \) and \( s_0 \sim s_1 \) then \( t_0 \sim t_1 \).
Semantics of baseline policy I

Initial values of H variables do not influence final values of L variables.

Let \( \bar{l} \) be the low variables and let \( s, t \) be states. Write \( s \sim t \) iff \( s \) and \( t \) agree on the low variables. Write \( s \rightarrow^* t \) to say the program yields \( t \) from \( s \).

Non-interference: if \( s_i \rightarrow^* t_i \) and \( s_0 \sim s_1 \) then \( t_0 \sim t_1 \).

Expressed in relational Hoare logic:
pre: \( A(\bar{l}) \)  post: \( A(\bar{l}) \)

where we define \( s_0, s_1 \models A(\bar{l}) \) iff \( s_0 \sim s_1 \)
Declassify what: delimited release

Release diagnosis but not doctor’s notes.

Release parity of secret: $l := \text{declass}(h \% 2)$ in Jif.

pre: $A(h \% 2) \land A(l)$  post: $A(l)$
Declassify what: delimited release

Release diagnosis but not doctor’s notes.

Release parity of secret: \( l := \text{declass}(h\%2) \) in Jif.

pre: \( A(h\%2) \land A(l) \)  \hspace{1cm} post: \( A(l) \)

Idea: “escape hatch” expression \( h\%2 \) is low, although \( h \) is high.
Declassify what: delimited release

Release diagnosis but not doctor’s notes.

Release parity of secret: \( l := \text{declass}(h \% 2) \) in Jif.

pre: \( A(h \% 2) \land A(l) \)  \hspace{1cm} post: \( A(l) \)

Idea: “escape hatch” expression \( h \% 2 \) is low, although \( h \) is high.

Beware laundering:  \hspace{1cm} pre: \( A(h \oplus h') \) \hspace{1cm} post: \( A(l) \)

\( h := h \oplus h'; \ l := \text{declass}(h \oplus h') \)
Declassify what: delimited release

Release diagnosis but not doctor’s notes.

Release parity of secret:  \( l := \text{declass}(h \% 2) \) in Jif.

pre: \( A(h \% 2) \land A(l) \)  post: \( A(l) \)

Idea: “escape hatch” expression \( h \% 2 \) is low, although \( h \) is high.

Beware laundering:  pre: \( A(h \text{ xor } h') \)  post: \( A(l) \)

\( h := h \text{ xor } h' \); \( l := \text{declass}(h \text{ xor } h') \)

\( h' := 0 \); \( l := \text{declass}(h \text{ xor } h') \)

Policy refers to initial state; ok if code can’t update \( h, h' \).
Declassify when: NI until

Release diagnosis only when log entry has been made and clerk/rep authenticated.

If \( s_i \rightarrow^* t_i \) and \( s_o \sim s_1 \) then either \( t_o \sim t_1 \) or else \( t_i \) satisfies the condition.

Or: the condition is true of one of the traces \( s_i \rightarrow \rightarrow \ldots \rightarrow t_i \)
Declassify when: NI until

Release diagnosis only when log entry has been made and clerk/rep authenticated.

If $s_i \rightarrow^* t_i$ and $s_0 \sim s_1$ then either $t_0 \sim t_1$ or else $t_i$ satisfies the condition.

Or: the condition is true of one of the traces $s_i \rightarrow\rightarrow\rightarrow \ldots \rightarrow t_i$

[Chong, Myers/04] label $H \rightsquigarrow^c L$ means NI until $c$

Weak property: not end-to-end. Anything can happen once condition is true.
Declassify where

Release diagnosis only when log entry has been made and clerk/rep authenticated.

—where in the code is there a declass and what conditions hold there?

Definitions:

A trace $\sigma$ is sequence of states resulting from assignments, with declass steps tagged as such.

$\text{purge}(\sigma)$ deletes the states resulting from H assignments.

$\sigma \sim \tau$ iff $\text{lowvis}(\text{purge}(\sigma)) = \text{lowvis}(\text{purge}(\tau))$ where $\text{lowvis}$ throws away H variables.
Semantics of baseline policy II

Recall $\sigma \sim \tau$ iff $\text{lowvis}(\text{purge}(\sigma)) = \text{lowvis}(\text{purge}(\tau))$

Non-interference with intermediate observations:

If $s \sim t$
and $\sigma, \tau$ are the corresponding complete traces
then $\sigma \sim \tau$. 
Gradual release

Define uncertainty about initial state, $\mathcal{U}(\sigma)$, to be
\[
\{ s \mid \exists \tau : \tau \sim \sigma \land \tau_0 = s \} 
\]
Gradual release

Define uncertainty about initial state, $\mathcal{U}(\sigma)$, to be
\[
\{ s \mid \exists \tau : \tau \sim \sigma \land \tau_o = s \}
\]

Note that $\mathcal{U}(\sigma t) \subseteq \mathcal{U}(\sigma)$ always.

Gradual release property: for all $\sigma, t$,
if $\sigma t$ is a trace (so last step went to $t$) then

♦ either $\mathcal{U}(\sigma t) = \mathcal{U}(\sigma)$
♦ or the step to $t$ is a declass
Gradual release examples

G.R.: for all $\sigma, t$, if $\sigma \cdot t$ is a trace then either $\mathcal{U}(\sigma \cdot t) = \mathcal{U}(\sigma)$ or the step to $t$ is a declass.
Gradual release examples

G.R.: for all $\sigma, t$, if $\sigma \ t$ is a trace then either $U(\sigma \ t) = U(\sigma)$ or the step to $t$ is a declass.

Secure: $l := \text{declass}(h)$
Gradual release examples

G.R.: for all $\sigma, t$, if $\sigma t$ is a trace then either $\mathcal{U}(\sigma t) = \mathcal{U}(\sigma)$ or the step to $t$ is a declass.

Secure: $l := \text{declass}(h)$

Insecure:

$h_1 := h_2; \ h_2 := 0; \ l := \text{declass}(h_2); \ h_2 := h_1; \ l := h_2$

Satisfies pre: $A(h_2)$ post: $A(l)$ but not G.R.
Gradual release examples

G.R.: for all $\sigma, t$, if $\sigma t$ is a trace then either $U(\sigma t) = U(\sigma)$ or the step to $t$ is a declass.

Secure: $l := \text{declass}(h)$

Insecure:

$h_1 := h_2; \ h_2 := 0; \ l := \text{declass}(h_2); \ h_2 := h_1; \ l := h_2$

Satisfies pre: $A(h_2)$ post: $A(l)$ but not G.R.

$l := h; \ loop$ satisfies G.R. because only terminating computations are considered.

But this can be fixed: treat termination as observable event and don’t discard divergent computations.
Conditional gradual release

Instead of, e.g., \( l := \text{declass}(\ldots) \), specify the declass code with a precondition \( P \land \phi \)
where \( P \) is state predicate, \( \phi \) is conjunction of agreements.

Idea: at every step, if attacker uncertainty decreases then it is a declass with policy \( P \land \phi \), and

- \( P \) is true at that step, and
- the step satisfies \( \text{pre: } P \land \phi \quad \text{post: } A(\overline{l}) \)
where \( \overline{l} \) are the program’s low variables.
Examples

\[ \text{if } h_1 \text{ then } l \leftarrow \text{declass}(h_2) \text{ is insecure: observing termination with } l \text{ unchanged implies } h_1 \text{ false, but no declassify step is observed.} \]
Examples

if \( h_1 \) then \( l := \text{declass}(h_2) \) is insecure: observing termination with \( l \) unchanged implies \( h_1 \) false, but no declassify step is observed.

\[ b := \text{declass}(h \neq 0); \text{if } b \text{ then } l := \text{declass}(h_1) \text{ else skip} \]

is ok for policy with \( A(h \neq 0) \) for first declass and \( A(h_1) \) for second.
Enforcing delimited release

Policy: set of escape-hatch expressions (Jif: those in declass expressions)

Typecheck the Denning conditions, but exempt declass statements and disallow them in H contexts (under H guards).

For each \( l := \text{declass}(E) \), \( E \) must be an escape-hatch (trivial syntactic match, could do better).

Disallow updates of H variables.
Enforcing gradual release

Policy: mark assignments that are allowed to declassify (in this talk, written $x := declass(y)$)

Typecheck the Denning conditions, but exempt declassify statements and disallow them in H contexts.
Enforcing gradual release

Policy: mark assignments that are allowed to declassify (in this talk, written $x := \text{declass}(y)$)

Typecheck the Denning conditions, but exempt declassify statements and disallow them in H contexts.

$l := \text{declass}(h); \quad l := h$ is satisfies G.R. (vs. in flow locks [Broberg,Sands] and other attempts), but is rejected
Enforcing C.G.R.

Policy: Labeling for baseline. Each marked declass has a “flowspec” pre: $P \land \phi$

♦ Type-check Denning conditions, exempting declass; no declass in H context; and no writes to $h$ in $\phi$

♦ For each declassification $C$, verify that $C$ satisfies pre: $P \land \phi$ post: $A(\bar{l})$ with $\bar{l}$ the variables writable by $C$, where $P \land \phi$ is the flowspec for $C$.

♦ For each declassification and flowspec $P \land \phi$, verify that $P$ is a valid assert preceding $C$. 
Results about C.G.R.

Thm: enforcement is sound

Simple way to treat escape-hatches semantically.

Security property has good properties [Sabelfeld, Sands’06 CSF Dimensions and Principles of Declassification]
Conclusion and related work

Knowledge-based formulation of information flow facilitates policies that stipulate where in the code releases are permitted, what is released there, and when can that code be executed.

[Cohen’78; Amtoft,Banerjee’04] logical formulation, $A(E)$ specifications

[Chong,Myers CCS’04] under what conditions can secrets be declassified

[Askarov,Sabelfeld S&P’07] (unconditional) gradual release and its use with key release
[B. Hicks et al PLAS’07] separate delim release policy from program

[Rushby’92, van der Meyden’07, etc] purging H events (and intransitive NI)
Future work

♦ “initial secrets” relative to when? start of transaction?
♦ region logic —agreements for heap objects
♦ declassification of data structures without cloning
♦ relational verifier
♦ integration of types, inference, verifier