Naïve and flexible declassification with scalable enforcement

Dave Naumann

Department of Computer Science
Stevens Institute of Technology

The Open Group, Real-Time Embedded Systems Forum
2 Feb 2009

Joint work with Anindya Banerjee and Stan Rosenberg (Oakland'08).

Partially supported by NSF grants CNS-0627338 and CNS-0708330.

Analysis tools exist for *information flow* in software, e.g.

- Sword4J (tracks taint in Java)
- Jif (security-typed Java, decentralized labels, etc)
- FlowCaml (subset of ocaml)
- SPARK/Ada
- dynamic control in runtime, e.g., taint mode in Perl
- dynamic control in OS

But static analysis sometimes too *conservative*,
dynamic analysis is *costly or incomplete*,
and *policy specifications are incomplete and restrictive.*

Analysis tools exist for *information flow* in software, e.g.

- Sword4J (tracks taint in Java)
- Jif (security-typed Java, decentralized labels, etc)
- FlowCaml (subset of ocaml)
- SPARK/Ada
- dynamic control in runtime, e.g., taint mode in Perl
- dynamic control in OS

But static analysis sometimes too *conservative*,
dynamic analysis is *costly or incomplete*,
and *policy specifications are incomplete and restrictive.*

Example policy: medical record release

- The patient's diagnosis is released to insurance representative, 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 office clerk and insurance rep.
- At the time of release, both clerk and representative should be users with valid credentials to act in their respective roles.

Conformant implementation?

```
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(); boolean t;
  PatientRecord pr := db.lookup(patID);
  if (subString("HIV",pr.diagnosis)) t := true;
  ir.id := pr.id;
  ir.diagnosis := pr.diagnosis;
  if (t) pr.id.setFont("italic");
  return ir; }
```

```
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?

Goal: automated static verifier that accepts the preceding code and rejects wrong versions.

Assuming:

- `sys.auth(...)` represents actual state of authentication
- authentication protocol is correct
- integrity of audit log
- platform uncorrupted (program means what it says)
- `release` executes atomically

But what does the policy even mean?

Conformant implementation?

Goal: automated static verifier that accepts the preceding code and rejects wrong versions.

Assuming:

- `sys.auth(...)` represents actual state of authentication
- authentication protocol is correct
- integrity of audit log
- platform uncorrupted (program means what it says)
- `release` executes atomically

But what does the policy even mean?

Outline of talk

- baseline policy and attacker model
- non-interference and delimited release —*what* flows
- gradual release —*where* in code
- our notion: conditioned 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 (i.e., platform/configuration not corrupted)
- 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 are visible; but not real time or other covert channels (cf. MILS architecture)

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 (i.e., platform/configuration not corrupted)
- 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 are visible; but not real time or other covert channels (cf. MILS architecture)

Semantics of baseline policy, vsn. 1

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 $s_0 \sim s_1$ and $s_0 \rightarrow^* t_0$ and $s_1 \rightarrow^* t_1$ then $t_0 \sim t_1$.

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

Using *agreement* defined by $(s_0, s_1) \models \mathbf{A}(\bar{l})$ iff $s_0 \sim s_1$ and considering two program runs.

Semantics of baseline policy, vsn. 1

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 $s_0 \sim s_1$ and $s_0 \rightarrow^* t_0$ and $s_1 \rightarrow^* t_1$ then $t_0 \sim t_1$.

Expressed in relational Hoare logic:
pre: $\mathbf{A}(\overline{l})$ post: $\mathbf{A}(\overline{l})$

Using *agreement* defined by $(s_0, s_1) \models \mathbf{A}(\overline{l})$ iff $s_0 \sim s_1$
and considering two program runs.

Declassify what: delimited release

Release diagnosis but not doctor's notes.

Release parity of secret: $l := \texttt{declass}(h\%2)$
pre: $\mathbf{A}(h\%2) \wedge \mathbf{A}(l)$ post: $\mathbf{A}(l)$

Idea: "escape hatch" expression $h\%2$ is low, although h is high.

Beware of laundering: pre: $\mathbf{A}(h \texttt{ xor } h')$ post: $\mathbf{A}(l)$
$h := h \texttt{ xor } h'; \; l := \texttt{declass}(h \texttt{ xor } h')$
Policy refers to initial state; ok as long as can't update h, h'.

Declassify what: delimited release

Release diagnosis but not doctor's notes.

Release parity of secret: $l := \texttt{declass}(h\%2)$
pre: $\mathbf{A}(h\%2) \wedge \mathbf{A}(l)$ post: $\mathbf{A}(l)$

Idea: "escape hatch" expression $h\%2$ is low, although h is high.

Beware of laundering: pre: $\mathbf{A}(h \texttt{ xor } h')$ post: $\mathbf{A}(l)$
$h := h \texttt{ xor } h'$; $l := \texttt{declass}(h \texttt{ xor } h')$
Policy refers to initial state; ok as long as can't update h, h'.

Declassify what: delimited release

Release diagnosis but not doctor's notes.

Release parity of secret: $l := \texttt{declass}(h\%2)$
pre: $\mathbf{A}(h\%2) \wedge \mathbf{A}(l)$ post: $\mathbf{A}(l)$

Idea: "escape hatch" expression $h\%2$ is low, although h is high.

Beware of laundering: pre: $\mathbf{A}(h \texttt{ xor } h')$ post: $\mathbf{A}(l)$
$h := h \texttt{ xor } h'$; $l := \texttt{declass}(h \texttt{ xor } h')$
Policy refers to initial state; ok as long as can't update h, h'.

Declassify what: delimited release

Release diagnosis but not doctor's notes.

Release parity of secret: $l := \texttt{declass}(h\%2)$
pre: $\mathbf{A}(h\%2) \wedge \mathbf{A}(l)$ post: $\mathbf{A}(l)$

Idea: "escape hatch" expression $h\%2$ is low, although h is high.

Beware of laundering: pre: $\mathbf{A}(h \texttt{ xor } h')$ post: $\mathbf{A}(l)$
$h := h \texttt{ xor } h'$; $l := \texttt{declass}(h \texttt{ xor } h')$
Policy refers to initial state; ok as long as can't update h, h'.

Declassify when: NI until

Release diagnosis only when log entry has been made and clerk and insurance representative authenticated.

If $s_i \longrightarrow^* t_i$ 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 $s_i \longrightarrow\!\!\!\!\!\rightarrow \ldots \longrightarrow t_i$

Label $H \rightsquigarrow^c L$ means NI until c holds [Chong,Myers'04].

Anything can happen thereafter; need stronger property.

Declassify when: NI until

Release diagnosis only when log entry has been made and clerk and insurance representative authenticated.

If $s_i \longrightarrow^* t_i$ 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 $s_i \longrightarrow \longrightarrow \ldots \longrightarrow t_i$

Label $H \rightsquigarrow^c L$ means NI until c holds [Chong,Myers'04].

Anything can happen thereafter; need stronger property.

Declassify where

Release diagnosis only when log entry has been made and clerk and insurance representative authenticated.

—where in the code is there a designated `declass`, and what conditions hold there? (cf. labels on downgraders)

Preliminary definitions:

A *trace* σ is sequence of states resulting from assignments.

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

$\sigma \sim \tau$ iff $lowvis(purge(\sigma)) = lowvis(purge(\tau))$
where *lowvis* throws away H variables.

Declassify where

Release diagnosis only when log entry has been made and clerk and insurance representative authenticated.

—where in the code is there a designated `declass`, and what conditions hold there? (cf. labels on downgraders)

Preliminary definitions:

A *trace* σ is sequence of states resulting from assignments.

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

$\sigma \sim \tau$ iff $lowvis(purge(\sigma)) = lowvis(purge(\tau))$
where $lowvis$ throws away H variables.

Semantics of baseline policy, vsn. 2

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

Non-interference with intermediate observations:
If $s \sim t$
and σ, τ are the corresponding complete traces
then $\sigma \sim \tau$.

Gradual release

Define $\mathcal{U}(\sigma)$, the uncertainty about initial state, w.r.t. observation σ, by $\mathcal{U}(\sigma) = \{s \mid \exists \tau : \tau \sim \sigma \wedge \tau_0 = s\}$

Note that $\mathcal{U}(\sigma\ t) \subseteq \mathcal{U}(\sigma)$ always. (trace σ followed by t)

Gradual Release property [Askarov,Sabelfeld'07]: for all σ, t, if $\sigma\ t$ is a trace then

- either $\mathcal{U}(\sigma\ t) = \mathcal{U}(\sigma)$
- or the step to t is a `declass`

To detect declass, consider runs with program counter.

Gradual release

Define $\mathcal{U}(\sigma)$, the uncertainty about initial state, w.r.t. observation σ, by $\mathcal{U}(\sigma) = \{s \mid \exists \tau : \tau \sim \sigma \wedge \tau_0 = s\}$

Note that $\mathcal{U}(\sigma\ t) \subseteq \mathcal{U}(\sigma)$ always. (trace σ followed by t)

Gradual Release property [Askarov,Sabelfeld'07]: for all σ, t, if $\sigma\ t$ is a trace then

- either $\mathcal{U}(\sigma\ t) = \mathcal{U}(\sigma)$
- or the step to t is a `declass`

To detect declass, consider runs with program counter.

Gradual release

Define $\mathcal{U}(\sigma)$, the uncertainty about initial state, w.r.t. observation σ, by $\mathcal{U}(\sigma) = \{s \mid \exists \tau : \tau \sim \sigma \wedge \tau_0 = s\}$

Note that $\mathcal{U}(\sigma\ t) \subseteq \mathcal{U}(\sigma)$ always. (trace σ followed by t)

Gradual Release property [Askarov,Sabelfeld'07]: for all σ, t, if $\sigma\ t$ is a trace then

- either $\mathcal{U}(\sigma\ t) = \mathcal{U}(\sigma)$
- or the step to t is a `declass`

To detect declass, consider runs with program counter.

Gradual release

Define $\mathcal{U}(\sigma)$, the uncertainty about initial state, w.r.t. observation σ, by $\mathcal{U}(\sigma) = \{s \mid \exists \tau : \tau \sim \sigma \wedge \tau_0 = s\}$

Note that $\mathcal{U}(\sigma\ t) \subseteq \mathcal{U}(\sigma)$ always. (trace σ followed by t)

Gradual Release property [Askarov,Sabelfeld'07]: for all σ, t, if $\sigma\ t$ is a trace then

- either $\mathcal{U}(\sigma\ t) = \mathcal{U}(\sigma)$
- or the step to t is a `declass`

To detect declass, consider runs with program counter.

Gradual release examples

Gradual release: for all σ, 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: $l := \texttt{declass}(h)$

Insecure: $h := h';\ h' := 0;\ l := \texttt{declass}(h');\ h' := h;\ l := h'$

But it does satisfy pre: $\mathbf{A}(h')$ post: $\mathbf{A}(l)$.

Gradual release examples

Gradual release: for all σ, 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: $l := \texttt{declass}(h)$

Insecure: $h := h';\ h' := 0;\ l := \texttt{declass}(h');\ h' := h;\ l := h'$

But it does satisfy pre: $\mathbf{A}(h')$ post: $\mathbf{A}(l)$.

Gradual release examples

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

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

Insecure: $h := h';\ h' := 0;\ l := \texttt{declass}(h');\ h' := h;\ l := h'$
But it does satisfy pre: $\mathbf{A}(h')$ post: $\mathbf{A}(l)$.

Conditioned gradual release

Specify each $l := \texttt{declass}(E)$ with a precondition $P \wedge \phi$ where P is state predicate, ϕ is conjunction of agreements.

Conditioned gradual release (CGR): at every step, if attacker uncertainty decreases then it is a declass step with policy $P \wedge \phi$, and

- P is true at that step, and
- the step satisfies pre: $P \wedge \phi$ post: $\mathbf{A}(\bar{l})$ where $\bar{l}$ are the program's low variables.

Small examples of CGR

$\texttt{if } h \neq 0 \texttt{ then } l := \texttt{declass}(h') \texttt{ else skip}$
is insecure: observing termination with l unchanged implies $h = 0$, but no declassify step is observed.

$b := \texttt{declass}(h \neq 0);\ \texttt{if } b \texttt{ then } l := \texttt{declass}(h') \texttt{ else skip}$
is secure for policy with $\mathbf{A}(h \neq 0)$ for first declass and $\mathbf{A}(h')$ for second.

Small examples of CGR

$\texttt{if } h \neq 0 \texttt{ then } l := \texttt{declass}(h') \texttt{ else skip}$
is insecure: observing termination with l unchanged implies $h = 0$, but no declassify step is observed.

$b := \texttt{declass}(h \neq 0);\ \texttt{if } b \texttt{ then } l := \texttt{declass}(h') \texttt{ else skip}$
is secure for policy with $\mathbf{A}(h \neq 0)$ for first declass and $\mathbf{A}(h')$ for second.

Enforcing delimited release

Policy: set of escape-hatch expressions. (Jif: those in `declass` expressions; better: separate from code [Hicks et al'06].)

Typecheck the no-read-up/write-down conditions,
but exempt declass statements and disallow them in H contexts (under H guards).

For each $l := \texttt{declass}(E)$, 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, e.g., by writing $x := \mathit{declass}(y)$

Typecheck the no-read-up/write-down conditions, but exempt declassify statements and disallow them in H contexts.

$l := \mathtt{declass}(h);\ l := h$ is secure for G.R., but is rejected.

Enforcing gradual release

Policy: mark assignments that are allowed to declassify, e.g., by writing $x := \mathit{declass}(y)$

Typecheck the no-read-up/write-down conditions, but exempt declassify statements and disallow them in H contexts.

$l := \texttt{declass}(h);\ l := h$ is secure for G.R., but is rejected.

Enforcing CGR

Policy: Labeling for baseline. And to each $l := \texttt{declass}(E)$ attach a flowspec, pre: $P \wedge \phi$

- Type-check no-read-up/write-down, exempting declass; no declass in H context; no updates of H variables (in ϕ).
- For each $l := \texttt{declass}(E)$ with flowspec $P \wedge \phi$, verify that it satisfies
  pre: $P \wedge \phi$ post: $\mathbf{A}(l)$
- Verify that P is a valid pre-assertion.

Type-checker + relational verifier + assertion verifier.

Enforcing CGR

Policy: Labeling for baseline. And to each $l := \texttt{declass}(E)$ attach a flowspec, pre: $P \wedge \phi$

- Type-check no-read-up/write-down, exempting declass; no declass in H context; no updates of H variables (in ϕ).
- For each $l := \texttt{declass}(E)$ with flowspec $P \wedge \phi$, verify that it satisfies
  pre: $P \wedge \phi$ post: $\mathbf{A}(l)$
- Verify that P is a valid pre-assertion.

Type-checker + relational verifier + assertion verifier.

What and when policies

Policy should be separate from code. ("Where" policy is indirect approach to "when", cf. intransitive NI.)

Schematic flowspec pre: $P \wedge \phi$ modif: x of type T
In P-states, can release, in any variable x, info bounded by ϕ.

Instead of rejecting bad direct flows $x' := E$, look for flowspec such that

- x' has type T
- $P' \wedge \phi'$ implies $\mathbf{A}(E)$
- P' is valid pre-assertion

(renaming schema variable x to x')

Flowspecs and heap regions

Example:

P : $pr.committed \wedge db.recent(pr) \wedge$
$sys.auth(b, \text{'book'}) \wedge sys.auth(r, \text{'rep'}) \wedge$
$log.contains(b.id, r.id, pr.id, pr.vsn, \text{'release'})$

ϕ : $\mathbf{A}(pr.diagnosis)$

modif: $ir.diagnosis$

diagnosis field is pointer to immutable string —but what if it was root of a data structure with mutable parts? e.g., a query result as collection object.

Subsequent H writes could be reachable/visible to L.

Flowspecs and heap regions

Example:

P : $pr.committed \wedge db.recent(pr) \wedge$
$sys.auth(b, \text{'book'}) \wedge sys.auth(r, \text{'rep'}) \wedge$
$log.contains(b.id, r.id, pr.id, pr.vsn, \text{'release'})$

ϕ : $\mathbf{A}(pr.diagnosis)$

modif: $ir.diagnosis$

diagnosis field is pointer to immutable string —but what if it was root of a data structure with mutable parts? e.g., a query result as collection object.

Subsequent H writes could be reachable/visible to L.

Region logic

Assertion language uses region expressions denoting sets of locations.

pre: $Owns(r, G)$ all access paths to objects in G go via r

$\mathbf{A}(G.*)$ agreement on all fields of objects in G

Mutable region expressions in modifies clause: reasoning about separation without reachability (facilitates automated deduction, cf. VCC for MS Hypervisor)

Region logic

Assertion language uses region expressions denoting sets of locations.

pre: $Owns(r, G)$ all access paths to objects in G go via r

$\mathbf{A}(G.*)$ agreement on all fields of objects in G

Mutable region expressions in modifies clause: reasoning about separation without reachability (facilitates automated deduction, cf. VCC for MS Hypervisor)

Summary

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.

Straightforward enforcement by typing, ordinary assertions, and *local relational verification*.
(Bonus: avoid conservativity of typing.)

Theorem: enforcement is sound

Security property avoids anomalies [Sabelfeld,Sands'05 Dimensions and Principles of Declassification].

[Banerjee,Naumann,Rosenberg S&P'08] this work
[Cohen'78; Amtoft,Banerjee'04] logical formulation, $\mathbf{A}(E)$ specifications
[Chong,Myers CCS'04] under what conditions can secrets be declassified
[Askarov,Sabelfeld S&P'07] (unconditional) gradual release and its use with crypto key release
[B.Hicks et al PLAS'06] separate delim release policy from program, substantial case study
[Rushby'92, van der Meyden'07, etc] purging H events; intransitive NI for coarse-grained policy
[Askarov, Sabelfeld PLAS'07] what and where, without when

Future work

- applications
- "initial secrets" relative to start of session; further: nondeducibility on strategies [Wittbold,Johnson; O'Niell,Clarkson,Chong]
- verifier for region logic [ECOOP'08, VSTTE'08]; decision procedure for regions
- verifier for relational region logic —PVS prototype first
- enforcement tool: integration of type inference/checking, assertion checker, relational verifier