Towards a Logical Account of Declassification

Anindya Banerjee
Kansas State University, Manhattan, KS, USA
ab@cis.ksu.edu

David A. Naumann Stan Rosenberg
Stevens Institute of Tech., Hoboken, NJ, USA
[naumann|srosenbe]@cs.stevens.edu

Abstract

Declassification is a vital ingredient for practical use of secure systems. Several recent efforts to formulate an end-to-end policy for declassification seem inconclusive and have focused on apparently different aspects. (e.g., what values are involved, where in the code declassification occurs, when declassification happens and who (which principal) releases information.) In this informal paper, we argue that key security goals addressed by the proposed notions can be expressed using assertions and auxiliary state (such as event history), building on a recently developed logic for noninterference that provides for local reasoning about the heap.

Categories and Subject Descriptors D.3.3 [*Software*]: Programming Languages—Language Constructs and Features; F.3.1 [*Theory of Computation*]: Logics and Meanings of Programs—Specifying and Verifying and Reasoning about Programs

General Terms Security, Languages, Verification.

Keywords aliasing, information flow, confidentiality.

1. Introduction

This paper argues for the use of relational program logic [6, 1] to specify information flow policies involving declassification as well as to verify compliance. The focus is on sequential, deterministic programs in a language like Java but it should be evident that the ideas are pertinent to concurrent programs and other programming languages.

Background. Confidentiality policy can be expressed by labeling variables with security levels. A program is *noninterferent* (for the policy) if every pair of computations, from a pair of initial states differing only in secrets, leads to final states with identical non-secrets. A number of works provide techniques for enforcement of noninterference for imperative and object-oriented programs. One approach treats security labels as non-standard types [23, 19]. By typing variable h as secret and l as low security, an evident rule disallows direct assignment of $l := h$ and additional constraints prevent implicit flows as in $\mathbf{if}\ h\ \mathbf{then}\ l := true$. The flow-insensitivities and value-abstractions that make type-checking fast also makes it reject many secure programs.

An alternative enforcement approach is to formulate security as a verification problem and use program logic [11, 9, 10]. Noninterference can be described by viewing one of the paired computations as acting on a renamed copy, say h', l', of the variables. For the two variables h, l, the equation $l = l'$ can be used as precondition and postcondition over state space h, h', l, l' to express the noninterference property of command S as a triple $\{l = l'\}\ S \parallel S'\ \{l = l'\}$ where S' is a renamed copy and $\parallel$ means disjoint parallel execution.

Java-like languages lack the $\parallel$ operator, but the idea can be realized in terms of a "relational Hoare logic" [6, 1].[1] In this paper we build on the work of Amtoft et al. [1] which addresses the key challenge for reasoning about object oriented programs—mutable data structure in the heap. Their logic appears much like conventional Hoare logic, but a triple $\{\phi\}S\{\psi\}$ involves pre- and post-conditions ϕ, ψ on pairs of program states and it is interpreted with respect to two executions of S. No explicit renaming is used. Instead, predicates can include what we now call *agreements*, of the form $l\ltimes$, which essentially means $l = l'$ (i.e., the two considered states agree on the value of l). Agreements can also involve *region* expressions which abstract the heap.

Leaving aside the heap, suppose S has low variables $l_0, \ldots, l_n$ and high variables $h_0, \ldots, h_k$, then for S to be noninterferent with respect to this labeling is equivalent to validity of the triple $\{\phi\}\ S\ \{\phi\}$ where ϕ is $l_0\ltimes \wedge \ldots \wedge l_n\ltimes$. It is also equivalent to validity of the triple $\{\phi\}\ S\ \{\phi'\}$ where ϕ' is the conjunction of agreements $l_i\ltimes$ for those l_i that are modifiable by S. Judgements in the logic actually take the form $\{\phi\}\ S\ \{\phi'\}\ [X]$ where the modifies set X specifies modifiable locations (modifiable object/field are described using region expressions). Compositional rules provide flow-sensitive reasoning and potentially incorporate reasoning about data (e.g., using decision procedures) and the modifies set facilitates the usual rule disallowing low writes in branches of high conditionals.

The logic of Amtoft et al [1] is decidable, using strongest-postcondition calculations, and is less conservative than type systems for Java-like languages [15, 4] owing to the use of regions. Modular reasoning about regions depends on a frame rule inspired by Separation Logic.

Declassification. A number of proposals have been made to extend enforcement techniques, especially type-based ones, to encompass declassification. Unfortunately, no compelling semantic property has emerged to provide an end-to-end meaning for policies with declassification. Extant proposals seem fragmented and offer complicated analyses for limited and sometimes obscure properties [21]. Several proposals which address the more difficult forms of declassification boil down to a "resetting" semantics in which an intransitive noninterference condition connects the initial state to the point where a declassification takes place, and then again imposes noninterference between that point and the final

[1] Or by embedding in standard program logic by composing the program with a renamed copy of itself [5, 22], a technique developed for reasoning about data abstraction in the 1970's [18] and recently extended to heap structure [16].

PLAS'07 June 14, 2007, San Diego, California, USA.

state (or next declassification). Some proposals offer policies that say declassification happens only under certain conditions or under the control of certain agents (but once it has happened, all bets are off). For lack of cogent semantics of declassification, the very well executed attempt by Sabelfeld and Sands [21] to make sense of the literature is only partly able to ground its informal principles in precise terms.

In this paper we borrow intuitions from existing proposals but formalize them as pre-post specifications in a relational Hoare logic. Without aiming to be exhaustive, we survey declassification examples from the recent literature and argue for decomposing the program, rather than the computations, and for specifying the components using agreements together with state predicates. The approach addresses several dimensions of declassification [21] including what information is released, where in the code it can occur, when it can occur and under whose authority.

Many of the recent proposals embed the security specification as part of the program, e.g., via a special declassification construct. In our approach, policy is expressed by pre-post specifications attached to subprograms intended to perform declassification. Triples in Hoare logic are intended to compose into proofs of complete programs. By contrast, our specifications for policy indicate an exemption from the baseline security policy and are not expected to compose. (But their validity can be checked using proof outlines and other means.) In Sect. 4 we consider detaching policy from code.

Our approach encompasses a range of policies in straightforward way. A number of anomalies found in previous proposals [21] are avoided. But a comprehensive assessment is left to future work.

Outline. Section 2 considers a range of declassification scenarios and introduces our approach in detail. Sect. 3 delves into richer examples and Sect. 4 puts our work in perspective.

2. Introducing Flowspecs

This section discusses some examples, leading to our proposal.

Electronic wallet. In the following example, from Sabelfeld and Myers [20], h is high and l, k are the low variables.

$$\textbf{if } h \geq k \textbf{ then } h := h - k; l := l + k \textbf{ else skip};$$

Informally, the declassification policy allows for a *partial release*: It is ok to reveal whether $h \geq k$, but nothing more about h must be revealed. Our specification is:

$$\{(h \geq k)\ltimes \wedge\, l\ltimes \wedge\, k\ltimes\} \_ \{l\ltimes \wedge\, k\ltimes\} \tag{1}$$

As in the case of pure noninterference, the postcondition contains agreements for all the low variables (or those in the modifies set, if present). Let us review the interpretation: a command S satisfies the specification provided that if it is run twice, from initial states that agree on l, on k, and on the value of expression $h \geq k$ —but not necessarily on the value of h— the final states agree on l and on k.

Sabelfeld and Myers [20] propose explicit $declassify$ expressions in code to serve as "escape hatches", marking the expressions whose (initial) values are to be released. For example, the guard condition in the code above would be written as $declassify(h \geq k, low)$.

Examples of this kind have been classified as "what" policies by Sabelfeld and Sands [21], who point out that their semantics is essentially noninterference in a general form allowing variation in the partial equivalence relations (PER) that describe distinctions considered visible. Thus the logical formulation is no surprise. It is more challenging to find satisfactory semantics for "when" policies, to which we turn next.

Sealed auctions. Consider two principals $Alice$ and Bob taking part in an auction. The protocol is as follows: $Alice$ and Bob place their bids; the system determines the higher bid and reveals its value and the identity of its bidder. The first policy says that neither bid influences the other. The second policy says that only the higher bid and its bidder are made public —and this of course only happens after bidding is concluded. The two policies address two separate concerns: the first, absence of cheating, is up to the point just before the outcome is determined. The second concern, absence of laundering, applies from that point through the action of publicizing the result.

The first concern can be addressed by standard type-checking. Let A, B be incomparable levels assigned to variables $aBid$ and $bBid$ respectively, where $bBid$ and $aBid$ are the bids placed by Bob (resp. $Alice$). The other program variables are $winBid$ for the winning bid and boolean $aliceWins$ to designate the winner; both are low. Then the $getBids$ phase should be noninterferent with respect to this labeling. The logical formulation uses agreements to express which variables and expressions are considered observable, or not. Policies involving lattices other than $low \leq high$ can be expressed using multiple triples; in this case, the $getBids$ code is subject to the two specifications $\{bBid\ltimes\} \_ \{bBid\ltimes\}$ (guarding $Alice$'s bid) and $\{aBid\ltimes\} \_ \{aBid\ltimes\}$ (guarding Bob's).

Regardless of how policy is expressed for $getBids$, this policy cannot be imposed on the $announceWinner$ code, which reveals the winning bid and the winner. Here is a possible specification:

$$\begin{array}{ll} pre: & (max(aBid, bBid))\ltimes \wedge (aBid \geq bBid)\ltimes \\ post: & aliceWins\ltimes \wedge winBid\ltimes \\ mod: & aliceWins, winBid \end{array} \tag{2}$$

Of course $winBid := max(aBid, bBid)$; $aliceWins := (aBid \geq bBid)$ satisfies specification (2). The modifies clause prevents, e.g., $aBid$ to be set to the winning bid.

In itself, the above specification expresses a "what" policy. But the second policy concern has the flavor of "where" or "when". The partial release should happen *after* bidding is concluded, and it should only happen in the code that is intended to announce the winner.

Our proposal is to express such policies by attaching one or more specifications to the designated subprogram(s). In this case, specification (2) should be attached to the code we have called $announceWinner$ and the noninterference specification should be attached to $getBids$.

For practical purposes, we propose that a *baseline policy* should be given by a conventional labeling. A "where" declassification policy can be specified by attaching one or more specifications to the designated subprograms. A specification used in this way is called a *flowspec*. The example has the form $getBids$; $announceWinner$ and the flowspec would be attached to $announceWinner$.

Policy should be enforced by type-checking as usual, but a fragment with an attached flowspec is exempt from type-checking —where the flowspec serves to designate a controlled violation of the baseline policy. Instead, the corresponding triple must be checked for each fragment and each of its attached flowspecs.

We would argue that absent a compelling end-to-end semantics for declassification, one might choose to content oneself with "pretty good security" achieved by these fragmentary checks. Even so, some healthiness conditions are needed for policies to be sensible. For example, a flowspec in the scope of a high guard condition would allow more leakage than it makes explicit. We disallow that. Before addressing the other healthiness conditions, we proceed to perhaps the most unique and powerful feature of flowspecs.

3. Connecting policies with requirements

The pure notion of "where" policy is that such policy designates a particular part of the code that is permitted to declassify. In intransitive noninterference, information may flow from high to low but only via a channel at a certain intermediate level. Such policies are incomplete reflections of the security goals they aim to achieve. For example, the intermediate level could be for certain principals trusted to assess the current sensitivity of data. The intermediate level could be assigned only to code in a certain component intended to sanitize the data in some way —or a "where" policy could directly designate that the component is allowed to declassify.

We would like to more fully express and enforce such requirements. This is where ordinary program specification and verification can play a role. A $sanitize$ routine could be proved to ensure the absence of certain SQL injection attacks in a query string, or more simply to erase certain sensitive fields of a medical patient record. Such subprograms can be exempted from the baseline security policy and can also be specified to execute only on condition that some prior event has occured, such as conclusion of the bidding phase of an auction.

Conditional release. It is possible for partial release to be conditional, using conditional expressions [20]. For example let E stand for ($\mathbf{if}\ pay > thresh\ \mathbf{then}\ h\ \mathbf{else}\ 0$), then $declassify(E, low)$ releases h only if adequate payment is made. The corresponding flowspec has precondition $E{\ltimes}$ and postcondition $result{\ltimes}$. The meaning of $E{\ltimes}$ is that for any two runs of a program the value of the conditional is the same. Note that it is possible that different branches of the conditional get executed in the two runs and yet yield the same value (i.e., in case $h = 0$).

An alternative specification might use disjunction in the precondition: $pay \leq thresh \vee h{\ltimes}$. A bit of care is needed when combining state predicates with agreements since the latter involve both of the pair of states. For the moment we shall be explicit and write $both(P)$, where P is a state predicate, to say that both states satisfy P. Thus we can distinguish between $both(P) \vee \phi$, which uses the boolean operator $\vee$ at the level of state-pairs, and $both(P \vee Q)$. Thus $both(P) \vee both(Q)$ is strictly stronger than $both(P \vee Q)$. The example can be written $both(pay \leq thresh) \vee h{\ltimes}$, so it is satisfied in a pair of initial states if either $pay > thresh$ is false in both or the value of h is the same in both. So the specification differs in meaning from the one using the conditional expression only when $h = 0$.

Sabelfeld and Myers suggest that conditional release can be extended to disjunctive policies, e.g. either h_1 or h_2 but not both can be released (these could be cards to be revealed in a round of a game). They do not propose a syntax. Our conjecture is that this is because their syntax is too closely tied to the program (the policy is determined by the conjunction of all the $declassify$ statements). The policy is easily written as a specification:

$$\{h_1{\ltimes} \vee h_2{\ltimes}\} \_ \{result{\ltimes}\} \quad .$$

Conjunctive preconditions. Static labeling is a strong and simple form of access control: an attacker is (assumed to be) unable to read or write fields labeled at higher levels. There have been some explorations of state-dependent labeling. In Flow Locks [7], a label is conditioned on a boolean variable —a ghost variable, in the verification lingo, subject to updates in program annotations. Pistoia et al [17] formulate indistinguishability in terms of the parts of state that are protected by certain permission checks.

If the currently enabled permissions are given by a ghost variable, say $SecCtx$, updated according to the semantics of Java stack inspection, then we can specify the following policy: h may be released but only if permission ph is enabled. The relevant code is subject to two flowspecs. One covers the situation where the permission is enabled:

$$\{both(ph \in SecCtx) \wedge h{\ltimes}\} \_ \{l{\ltimes}\}\ [l]$$

This imposes no constraint on program executions where ph is initially not enabled, for which a second flowspec is used:

$$\{both(ph \notin SecCtx)\} \_ \{l{\ltimes}\}\ [l]$$

The modifies set, $[l]$, spares us from enumerating, in the postcondition, agreements for all low locations.

A feature of this example is that the preconditions can be expressed as conjunctions of the form $both(P) \wedge \phi$ where ϕ consists of agreements. For such a policy, it is straightforward to express that it does not admit attacks through omission of possible executions: we insist that the ordinary state predicate, Q, obtained as the disjunction of the state predicates P in preconditions, be a valid assertion at the beginning of the fragment to which the flowspec is attached. That is, the flowspec should cover all cases — a healthiness condition for the flowspec.

Release in multiple steps. Chong and Myers [8] use types to express policies that allow declassifications only after several conditions have been true in succession. They do not give an example with multiple steps so we describe our own. In fact, since in this paper we use assertions rather than temporal logic, we cannot directly express sequences of events. We consider an example with this flavor, and note that the event history is naturally encoded in the program state, just as in the auction example the end of bidding is a definite state in the program (indeed control point, in our toy example).

Consider a record for information about a medical patient. A bookkeeper needs to release the patient's information to an insurance representative. This is governed by the following policy:

- The patient's diagnosis is released, but not the doctor's notes (both are normally secret).
- The version of the record to be released should be the most recent one.
- The record should be in "committed" state. The database contains some versions that record saved test info; a committed record reflects a doctor's firm diagnosis.
- 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 bookkeeper and representative should be users with valid credentials to act in their respective roles. (We want to enforce this policy in the code, but one might also include in the log entry a timestamp, to facilitate *post hoc* correlation with the authentication logs to check that the bookkeeper and representative were indeed authorized.)

The example is illustrative, but not realistic. For example, we do not mention integrity of the logs, nor do we consider a realistic system of roles, or restrict to representatives of the patient's insurance company.

For purposes of the example, security level L is associated with information for which at least the insurance company is permitted access, and H is associated with private patient information and clinic-internal information. The clinic's database contains records of this form:

```
class PatientRecord {
  int id;   boolean committed;  int vsn;
  String<H> diagnosis;  String<H> notes;  }
```

A similar record is provided to insurance representatives. Note that L fields are unmarked.

```
class InsRecord {
  int id;  String diagnosis; }
```

Before formalizing the policy we give a conforming implementation.

```
Object release(Database db, int patID,
               Bookkeeper b, InsRep r)
  pre: sys.auth(b,"book") && sys.auth(r,"rep")
{
  InsRecord ir = new InsRecord();
  PatientRecord pr = db.lookup(patID);
  if (pr != null && pr.committed) {
    log.append(b.id, r.id,
               pr.id, pr.vsn, "release");
    ir.id = pr.id;
    ir.diagnosis = pr.diagnosis;
    return ir;
  } else { return new Msg("not available"); }
}
```

Note that the parameters and local variables are all low (unmarked). The only high things are certain fields of patient records. With this labeling, the program type-checks except for the assignment[2]

```
    ir.diagnosis = pr.diagnosis;
```

Consider the following flowspec to be attached to this assignment.[3] For clarity we omit "$both$" around the state predicates.

$$
\begin{array}{ll}
pre: & pr.diagnosis{\ltimes} \wedge pr.committed \\
& \wedge sys.auth(b, \text{'book'}) \wedge sys.auth(r, \text{'rep'}) \\
& \wedge db.recent(pr) \\
& \wedge log.contains(b.id, r.id, pr.id, pr.vsn, \text{'release'}) \\
post: & ir.diagnosis{\ltimes} \\
mod: & ir.diagnosis
\end{array}
$$

Predicate $db.recent(pr)$ is assumed to express that pr is the most recent patient record.

The precondition $pr.diagnosis{\ltimes}$ allows partial release of the patient record. The other preconditions express the conditions under which this release is permitted.

This policy satisfies the healthiness conditions. The flowspec is not inside a high conditional. Its postcondition covers the modified locations. The state predicate in its precondition covers the possible states just before the assignment to `ir.diagnosis`, as we now argue in detail. The conjunct $pr.committed$ holds owing to the guard condition. The conjuncts $sys.auth(b, \text{'book'})$ and $sys.auth(r, \text{'rep'})$ are preconditions to the method —its calls must therefore be verified for these conditions. Recency should be a consequence of the specification of $lookup$. (This would get more complicated if we considered concurrent access to the database; the policy is perhaps too strong on this point.) Presence of the log entry is ensured by the call to $append$.

It would be more difficult to express the policy as a flowspec to be attached to the entire method body: First, the postcondition would need agreements for all low writes; second, we would not be able to refer to the log event as having occured, nor be sure the latest record is committed.

One benefit of using specifications as opposed to using declassify expressions in code is that the latter is limited to conditions that make sense in code whereas the former can refer to ghost state like message histories. In our example, $sys.auth$ could well refer to a system data structure. But one can also imagine a scenario where the code runs on a resource-limited device that does not use persistent state to track the state of the authentication protocols; then sys could be a ghost variable for reasoning about the protocol state.

4. Discussion

Our approach is to begin by labeling the program's interfaces using lattice levels as usual. Many methods in a program are likely to satisfy the baseline noninterference policy with respect to these levels and even to be accepted by a security type checker. For code subject to a declassification policy (and thus exempt from the baseline policy), we attach flowspecs and, instead of type-checking the code, prove the validity of the flowspecs in an underlying logic.

Flowspecs themselves must be subject to certain healthiness conditions.

- A flowspec should not occur in the scope of a high guard condition.
- The postcondition of a flowspec should consist of agreements for all locations in the modifies clause. (A practical concrete syntax would be streamlined accordingly.)
- The flowspecs attached to a particular declassification must be exhaustive in that their preconditions should cover all possible cases (which can depend on checking ordinary program assertions as in the clinic example).
- The declassification code should satisfy the flowspec.

There are analyses that can check whether the program releases more than the flowspec explicitly permits [3]; such situation is subject to laundering attacks, so the code and/or policy need to be refined.

In some sense, the proposal closest to ours is that of Chong and Myers [8] who address "where" and "when" policies by labeling variables with security types that refer to a sequence of conditions and levels through which downgrading is allowed to pass.[4] They propose to enforce their policies using a type system augmented with some means to reason about "conditions". The means most well understood and supported by tools is assertion-based verification. Typical conditions might refer to log entries or other data already present in the program state. To refer to event occurrences, ghost variables can be used. Use of such ghost variables/fields depends on properly annotating the program with ghost updates of course, e.g., to correctly track what events have occurred, but this is standard verification methodology.

The "who" dimension of policies is also important, e.g., an official may be authorized to release confidential records in a financial database to law enforcement officials exhibiting a subpoena [13]. The fact that authentication checks have been made is likely to be encoded in the program state and if necessary it too can be modeled with ghost variables. As in many such examples, (e.g., the password example [8] where release depends on a condition that means "only trusted code is running") the idea is that certain programs have been validated by various means, with respect to policies that are not formalized as part of the declassification policy per se.

It is desirable to separate policy from implementation. Just as the labeling of variables is somewhat separate from the code, one could imagine partial release policies being expressed us-

[2] Systems like Jif do some amount of inference for labels of internal variables etc. In that case, this example would be rejected but it might not be as clear which sub-program is offensive.

[3] We abuse the notation of our actual logic, which does not include agreements for heap-dependent expressions like $pr.diagnosis$. Instead, the desired condition would be written $\langle pr \rangle.diagnosis{\ltimes}$ using a singleton region.

[4] As indicated earlier, it is not clear to the authors the extent to which declassification levels per se are an artifice in treatments of declassification. The real-world policies seem to involve declassification being controlled by suitable authorities and via specific channels and operations, rather than intrinsically meaningful intermediate security levels.

ing an augmented labeling that designates levels for certain "escape hatch" expressions, overriding the level given by usual typing rules; e.g., $h \geq k$ could be declared low despite the join of its variable levels being high. This is explored by Hicks et al [12]. Similarly, we believe that many richer policies can be expressed using schematic flowspecs. In our example, the assignment $ir.diagnosis := pr.diagnosis$ makes sense in any context where ir and pr are declared. Moreover, the pre- and post-conditions refer to fields of these objects and to global data structures (the log and the authentication system). So it can be read as a schematic policy, applicable to any assignments from fields of `PatientRecord` objects to `InsRecord` objects (in any context where the globals are visible). Such a schema could be used to automatically exempt matching subprograms from type-checking and at the same time impose proof obligations on each match. As pointed out by a reviewer, this idea needs to be developed in such a way that the semantics of a policy does not change as the result of equivalence-preserving transformations of the program.

Other related work. The Jif tool offers, for sequential Java, rich information policies including declassification and much more [15].

Mantel and Sands [13] give a bisimulation condition for intransitive noninterference but it is unclear whether this condition is suitable for sequential code.

An important recent advance is the security condition called gradual release [2]. In contrast with previous language-based notions of intransitive noninterference, this avoids state-resetting and associated anomalies, by using a formulation in terms of attacker knowledge. Release events are marked in the code and computation steps are not allowed to increase attacker knowledge (narrowing the range of possible values for the secret) except for release steps. The paper also considers encryption and key release which are not pertinent here.

Having compared their type system for gradual release with what we propose, we conjecture that our regime enforces gradual release (we do not expect major problems in adapting their definitions to encompass our richer language). Although gradual release is a "where" policy, Askarov and Sabelfeld suggest that it can be combined with "what" by imposing a partial release condition on the step taken by each release event. This seems plausible because what is released can be overapproximated using the techniques of Banerjee et al [3] and moreover we plan to push that further by incorporating state-based conditions on release steps, resulting in an end-to-end semantics for flowspecs. It may also be fruitful to make knowledge explicit using a modal operator in assertions [14]. Elsewhere we will also formalize flowspecs in the setting of a relational logic that extends that of Amtoft et al. [1] with a full assertion language.

Acknowledgement. A number of people have given helpful feedback on the work, including Torbin Amtoft, Tamara Rezk, Alejandro Russo, Andrei Sabelfeld, and Steve Zdancewic. The anonymous PLAS reviews were helpful and insightful.

References

[1] T. Amtoft, S. Bandhakavi, and A. Banerjee. A logic for information flow in object-oriented programs. In *POPL*, 2006. Extended version available as KSU CIS-TR-2005-1.

[2] A. Askarov and A. Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *2007 IEEE Symposium on Security and Privacy*. To appear.

[3] A. Banerjee, R. Giacobazzi, and I. Mastroeni. What you lose is what you leak: Information leakage in declassification policies. In *MFPS*, 2007.

[4] A. Banerjee and D. A. Naumann. Stack-based access control for secure information flow. *Journal of Functional Programming*, 15(2):131–177, 2005. Special issue on Language Based Security.

[5] G. Barthe, P. R. D'Argenio, and T. Rezk. Secure information flow by self-composition. In *CSFW*, 2004.

[6] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *POPL*, 2004.

[7] N. Broberg and D. Sands. Flow locks. In *ESOP*, 2006.

[8] S. Chong and A. C. Myers. Security policies for downgrading. In *CCS*, 2004.

[9] E. S. Cohen. Information transmission in sequential programs. In A. K. J. Richard A. DeMillo, David P. Dobkin and R. J. Lipton, editors, *Foundations of Secure Computation*, pages 297–335. Academic Press, 1978.

[10] A. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In *SPC*, 2005.

[11] D. E. Denning. *Cryptography and Data Security*. Addison-Wesley, 1982.

[12] B. Hicks, D. King, P. McDaniel, and M. Hicks. Trusted declassification: high-level policy for a security-typed language. In *PLAS*, 2006.

[13] H. Mantel and D. Sands. Controlled declassification based on intransitive noninterference. In *APLAS*, 2004.

[14] C. Morgan. The shadow knows: Refinement of ignorance in sequential programs. In *Mathematics of Program Construction*, volume 4014 of *LNCS*, 2006.

[15] A. C. Myers. JFlow: Practical mostly-static information flow control. In *POPL*, 1999.

[16] D. A. Naumann. From coupling relations to mated invariants for secure information flow and data abstraction. In *ESORICS*, 2006.

[17] M. Pistoia, A. Banerjee, and D. A. Naumann. Beyond stack inspection: A unified access-control and information-flow security model. In *2007 IEEE Symposium on Security and Privacy*. To appear.

[18] J. C. Reynolds. *The Craft of Programming*. Prentice-Hall, 1981.

[19] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, Jan. 2003.

[20] A. Sabelfeld and A. C. Myers. A model for delimited information release. In *ISSS*, 2004.

[21] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *CSFW*, 2005. Full version to appear in *Journal of Computer Security*, 2007.

[22] T. Terauchi and A. Aiken. Secure information flow as a safety problem. In *SAS*, 2005.

[23] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.