Improving the Semantics of Imperfect Security

Niklas Broberg and David Sands

Chalmers University of Technology, Sweden

Information flow policies that evolve over time (including, for example, declassification) are widely recognised as an essential ingredient in useable information flow control system. In previous work ([BS06a, BS06b]) we have shown one approach to such policies, *flow locks*, which is a very general and flexible system capable of encoding many other proposed approaches.

However, any such policy approach is only useful if we have a precise specification – a semantic model – of what we are trying to enforce. A semantic model gives us insight into what a policy actually guarantees, and defines the precise goals of any enforcement mechanism. Unfortunately, semantic models of declassification can be both inaccurate and difficult to understand. This was definitely the case for the flow locks system as presented in [BS06a, BS06b], and we have found that the main problem is one common to most proposed models to date. We will start by discussing the problem in general, and then go on to sketch its solution for the flow locks system specifically.

The Flow Sensitivity Problem. The most commonly used semantic definition of secure information flow defines security by comparing any two runs of a system in environments that only differ in their secrets. A system is secure or *noninterfering* if any two such runs are indistinguishable to an attacker. Many semantic models for declassification are built from adaptations of such a two-run noninterference condition.

Such adaptations are problematic for a number of reasons, but the most fundamental problem is that they lack a formal and intuitive attacker model. An attacker would not observe two runs and try to deduce something from their differences, which is what a two-run model inherently claims.

There are also technical problems arising from the use of a two-run system. Consider the first point in a run at which a declassification occurs. From this point onwards, two runs may very well produce different observable outputs. A semantic model for declassification must constrain the difference at the declassification point in some way, and further impose some constraint on the remainder of the computation. The prevailing approach (e.g. [MS04, EP05, EP03, AB05, Dam06, MR07, BCR08, LM08]) to give meaning to declassification is to reset the environments of the systems so as to restore the low-equivalence of environments at the point after a declassification. We refer to this as the *resetting approach* to declassification semantics.

The down-side of the resetting approach is that it is *flow insensitive*. This implies that the security of a program P containing a reachable subprogram Q requires that Q be secure independently of P. Another instance of the problem is that dead code can be viewed as semantically significant, so that a program will be rejected because of some insecure dead code. Note that flow insensitivity might be a perfectly reasonable property for a particular *enforcement* mechanism such as a type system – but in a sequential setting it has no place as a fundamental semantic requirement.

A Knowledge-based Approach. Comparing two runs is fairly intuitive for standard noninterference, but in the presence of policy changes such as the opening or closing of locks (in our work) or declassification (in other work) it can be hard to see how the semantic definition really relates to what we can say about an attacker.

One recent alternative to defining the meaning of declassification is Gradual Release [AS07], which uses a more explicit attacker model whereby one reasons about what an attacker learns about the initial inputs to a system as computation progresses.

The basic idea can be explained when considering the simple case of noninterference between an initial memory state, which is considered secret, and public outputs. The model assumes that the attacker knows the program itself P.

Suppose that the attacker has observed some (possibly empty) trace of public outputs t. In such a case the attacker can, at best, deduce that the possible initial state is one of the following: $K_1 = \{N \mid \text{Running } P \text{ on } N \text{ can yield trace } t\}$. Now suppose that after observing t the attacker observes the further output u. Then the attacker knowledge is $K_2 = \{N \mid \text{Running } P \text{ on } N \text{ can yield trace } t \text{ followed by } u\}$. For the program to be considered noninterfering, in all such cases we must have $K_1 = K_2$. The gradual release idea weakens noninterference by allowing $K_1 \neq K_2$, but only when u is explicitly labelled as a declassification event.

This style of definition is the key to our new flow lock semantics. The core challenge is then to determine what part of the knowledge must remain constant on observing the output u by viewing the trace from the perspective of the lock-state in effect at that time.

Flow locks: the basic idea. Flow locks are a simple mechanism for interfacing between information flow policies given to data, and the code that processes that data. Security policies are associated with the storage locations in a program. In general a *policy* p is a set of *clauses*, where each clause of the form $\Sigma \Rightarrow \alpha$ states the circumstances (Σ) under which actor α may view the data governed by this policy. Σ is a set of locks, and for α to view the data all the locks in Σ must be open.

A lock is a special variable in the sense that the only interaction between the program and the lock is via the instructions to *open* or *close* the lock. In this way locks can be seen as a purely compile-time entity used to specify the information flow policy. We say that x *is visible to* α *at* Δ if the policy of x contains $\Sigma \Rightarrow \alpha$ for some $\Sigma \subseteq \Delta$.

One aim of the flow locks approach is to provide a general language into which a variety of information flow mechanisms can be encoded, to let flow locks serve as a unifying framework for various policy mechanisms. As a simple example of such an encoding from [BS06a], consider a system with data marked with "high" or "low", and a single statement $\ell := \mathit{declassify}(h)$ taking high data in h and downgrading it to low variable ℓ. We can encode this using flow locks by letting low variables be marked with $\{\mathit{high}; \mathit{low}\}$ and high variables with $\{\mathit{Decl} \Rightarrow \mathit{high}; \mathit{low}\}$. The statement $\ell := \mathit{declassify}(h)$ can then be encoded with the following sequence of statements: `open` $\mathit{Decl}; \ell := h;$ `close` Decl.

In common with many of the approaches cited above, the previous semantic security model for flow locks is defined using a resetting approach. All the problems discussed above are thus manifested in the old flow locks security model – no intuitive attacker model leading to an imperfect notion of knowledge gained, and a flow insensitive notion of security that rules out many perfectly secure programs for purely technical reasons.

A new semantics for Flow Locks. First we need a suitable attacker model. The model given in [AS07] is very simple, to match the simple declassification mechanism they consider. For flow locks we need a slightly more complex model, both since we deal with multiple actors, but also since flows are more fine-grained, so that a secret may be declassified for an actor through a series of steps.

To verify that a program is allowed, we need to validate the flows at each "level" that a secret may flow to, where a level corresponds to a certain set of locks guarding a location from a given actor. We note that these levels correspond to the points in the lattice $Actors \times \mathcal{P}(Locks)$, which leads us to our formal attacker model: An attacker A is a pair of an actor α and a set of locks Δ, formally $A = (\alpha, \Delta) \in Actors \times \mathcal{P}(Locks)$. We refer to the lockstate component of an attacker as his *capability*, and say that a location x is visible to $A = (\alpha, \Delta)$ iff x is visible to α at Δ.

The flows that we need to validate are the assignments, so each assignment must be registered in the "trace" that the program generates when run. An output u in the trace is visible to attacker A if the variable being assigned to is, and an *A-observable trace* is the restriction of a full trace to only include the outputs visible to A.

To reason about attacker knowledge we also need to be able to focus on the parts of a memory which are visible to a given attacker. Given an attacker A, we say a memory L is A-low if $dom(L) = \{x \mid A \text{ can see } x\}$. We say that two memories M and N are A-equivalent, written $M \sim_A N$ if their A-low projections are identical – i.e. they agree on all variables that A can see. The knowledge gained by an attacker A from observing a sequence of outputs $\vec{w}$ of a program c starting with a A-low memory L is then defined to be the set of all possible starting memories that could have lead to that observation:

$$k_A(\vec{w}, c, L) = \{M | M \sim_A L, \text{running } c \text{ on } M \text{ can yield A-visible trace } \vec{w}\}$$

Intuitively, for a program to be flow lock secure we must consider the perspective of each possible attacker A, and how his knowledge of the initial memory evolves as he observes successive outputs. The requirement for each output thus observed is that knowledge of the initial memory only increases if the attacker's inherent capabilities are weaker than the program lockstate in effect at the time of the output. The intuition here is that an attacker whose capability includes the program lock state in effect should already be able to see the locations used when computing the value that is output. Thus no knowledge should be gained by such an attacker.

For convenience we introduce the notion of a *run*, which is just an output trace together with the lockstate in effect at the time of the last output in the sequence:

$$\begin{aligned} Run_A(\Sigma, c, L) = \{(\vec{w}w, \Omega) | M \sim_A L, \\ &\text{running } c \text{ with starting memory } M \text{ and starting lockstate } \Sigma \text{ yields} \\ &A\text{-visible output trace } \vec{w}w \text{ where } \Omega \text{ is the lockstate in effect at output } w\} \end{aligned}$$

Finally we can define our security requirement in terms of runs. A program c is said to be Σ-flow lock secure, written $FLS(\Sigma, c)$ iff for all attackers $A = (\alpha, \Delta)$, all A-low memories L, and all runs $(\vec{w}w, \Omega) \in Run_A(\Sigma, c, L)$ such that $\Omega \subseteq \Delta$ we have

$$k_A(\vec{w}w, c, L) = k_A(\vec{w}, c, L)$$

This definition directly captures the intuition that we started out with. An attacker whose capabilities includes the current lockstate in effect at the time of the output should learn nothing new when observing that output. Attackers who do not fulfill this criterion have no constraint on what they may learn at this step. But note that this cannot lead to unchecked flows because we quantify over *all* attackers including, in particular, those with sufficient capabilities.

Further reading. This extended abstract accompanies an invited talk, and what we describe here is still work in progress. The slides and latest version of a full paper are available at: `http://www.cs.chalmers.se/~dave/ARSPA-WITS09`

References

[AB05] Almeida Matos, A., Boudol, G.: On declassification and the non-disclosure policy. In: Proc. IEEE Computer Security Foundations Workshop, June 2005, pp. 226–240 (2005)

[AS07] Askarov, A., Sabelfeld, A.: Gradual release: Unifying declassification, encryption and key release policies. In: Proc. IEEE Symp. on Security and Privacy, May 2007, pp. 207–221 (2007)

[BCR08] Barthe, G., Cavadini, S., Rezk, T.: Tractable enforcement of declassification policies. In: Proc. IEEE Computer Security Foundations Symposium (2008)

[BS06a] Broberg, N., Sands, D.: Flow locks: Towards a core calculus for dynamic flow policies. In: Sestoft, P. (ed.) ESOP 2006. LNCS, vol. 3924, pp. 180–196. Springer, Heidelberg (2006)

[BS06b] Broberg, N., Sands, D.: Flow locks: Towards a core calculus for dynamic flow policies. Technical report, Chalmers University of Technology and Göteborgs University (May 2006); Extended version of [BS06a]

[Dam06] Dam, M.: Decidability and proof systems for language-based noninterference relations. In: Proc. ACM Symp. on Principles of Programming Languages (2006)

[EP03] Echahed, R., Prost, F.: Handling harmless interference. Technical Report 82, Laboratoire Leibniz, IMAG (June 2003)

[EP05] Echahed, R., Prost, F.: Security policy in a declarative style. In: Proceedings of the 7th International Conference on Principles and Practice of Declarative Programming (PPDP 2005), Lisboa, Portugal (July 2005)

[LM08] Lux, A., Mantel, H.: Who can declassify? In: Preproceedings of the Workshop on Formal Aspects in Security and Trust (FAST) (2008)

[MR07] Mantel, H., Reinhard, A.: Controlling the what and where of declassification in language-based security. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 141–156. Springer, Heidelberg (2007)

[MS04] Mantel, H., Sands, D.: Controlled declassification based on intransitive noninterference. In: Chin, W.-N. (ed.) APLAS 2004. LNCS, vol. 3302, pp. 129–145. Springer, Heidelberg (2004)