Flow-Sensitive Semantics for Dynamic Information Flow Policies

Niklas Broberg
Department of Computer Science and Engineering,
Chalmers University of Technology and University of Gothenburg
Göteborg, Sweden
d00nibro@chalmers.se

David Sands
Department of Computer Science and Engineering,
Chalmers University of Technology
Göteborg, Sweden
dave@chalmers.se

Abstract

Dynamic information flow policies, such as declassification, are essential for practically useful information flow control systems. However, most systems proposed to date that handle dynamic information flow policies suffer from a common drawback. They build on semantic models of security which are inherently flow insensitive, which means that many simple intuitively secure programs will be considered insecure.

In this paper we address this problem in the context of a particular system, flow locks. We provide a new flow sensitive semantics for flow locks based on a knowledge-style definition (following Askarov and Sabelfeld), in which the knowledge gained by an actor observing a program run is constrained according to the flow locks which are open at the time each observation is made. We demonstrate the applicability of the definition in a soundness proof for a simple flow lock type system. We also show how other systems can be encoded using flow locks, as an easy means to provide these systems with flow sensitive semantics.

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Langauges

General Terms Languages, Security

Keywords Information Flow Control, Declassification, Security Type System

1. Introduction

Information flow policies that evolve over time (including, for example, declassification) are widely recognised as an essential ingredient in useable information flow control systems.

PLAS '09 June 15, Dublin, Ireland.

Reprinted from PLAS '09, ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security – **including appendix which does not appear in the published version of the paper.**, June 15, Dublin, Ireland., pp. 1–16.

Such policies are 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 – in particular those that try to specify more that just *what* is declassified – can be both inaccurate and difficult to understand.

We address this problem for one specific but rather flexible information flow policy approach, *flow locks*.

The Flow Sensitivity Problem The most commonly used semantic definition of secure information flow – at least in the language-based setting – involves the comparison of two runs of a system. The idea is to define security by comparing any two runs of a system in environments that only differ in their secrets (such environments are usually referred to as being *low equivalent*). A system is secure or *noninterfering* if any two such runs are indistinguishable to an attacker. These "two run" formulations relate to the classical notion of *unwinding* in [GM82].

Many semantic models for declassification – in particular those which have a "where" or "when" dimension [SS05] – are built from adaptations of such a two-run noninterference condition. [1]

Such adaptations are problematic. 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 declassification semantics must constrain the difference at the declassification point in some way (this is specific to the particular flavour of declassification at hand), and further impose some constraint on the remainder of the computation. So what constraint should be placed on the remainder of the computation? The prevailing approach to give meaning to declassification (e.g. [MS04, EP05, EP03, AB05, Dam06, MR07, BCR08, LM08]) is to reset the environments

[1] For the purposes of this paper it is useful to view declassification as a particular instance of a dynamic information flow policy in which the information flow policy becomes increasingly liberal as computation proceeds.

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. For example, consider the program

$$\text{declassify } h \text{ in } \{\ell := h\}; \ell := h$$

where h is a high security variable and ℓ is low. In the semantics of e.g. [BCR08] this would be deemed insecure because of the insecure subprogram $\ell := h$ – even though in all runs this subprogram will behave equivalently to the obviously secure program $\ell := \ell$. Similar examples can be constructed for all of the approaches cited above. 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.

The resetting approach is not without merits though. In particular it is able to handle shared-variable concurrency in a compositional way [MS04, AB05]. However, the use of resetting for compositionality and its use for giving a semantics to declassification are orthogonal, and the flow insensitivity problem carries over to those parts of the environment which are not shared across threads.

Overview In this paper we tackle the problem of providing a semantics for "dynamic"[2] information flow policies for one particular approach, *flow locks*. Flow locks (reviewed in Section 2) were introduced with the intention of providing a core calculus for expressing dynamic flow policies. We can encode a wide range of declassification mechanisms using flow locks, which we have shown in [BS06a, BS06b].

The earlier semantic model for flow locks suffers from the flow insensitivity problem described above. Perhaps due to its generality it is also overly complex and unintuitive. The key to recovering flow sensitivity and to drastically simplifying the semantics is to follow the lead of Askarov and Sabelfeld [AS07] who move away from a "two run" view of security semantics, and focus instead on how an explicit representation of the attacker's knowledge evolves as computation proceeds. This approach is reviewed in Section 2.

Using this approach we craft our new semantics (Section 3) and discuss some of the basic properties of the definition (Section 4) from the declassification perspective [SS05].

We go on to show that the definition is useable by applying it to a concrete instance and a simple type system for flow lock security (Section 5).

[2] For the purposes of this paper we use dynamic to refer to a policy which varies at runtime. Other notions of dynamic policy not considered in this work include, for example, runtime principals [TZ04] or labels [ZM07].

Finally we discuss encodings of other systems, and in particular we show (Section 6) that Askarov and Sabelfeld's basic *gradual release* property is soundly and completely represented by the flow locks encoding of simple declassification [BS06a].

2. Preliminaries

In this section we review the basic flow locks idea, some of the issues with its previous semantic model, and outline the knowledge-based alternative style of semantics that we will use to provide a new semantic model.

Flow locks: the basic idea Suppose we have a program which deals with two *actors*, a vendor and a customer. The program has access to the vendor's secret data – a software activation key – which should not be permitted to flow to the customer unless the customer has paid for the software. To model the payment act we have a special boolean flag called a *lock*. Let us call this particular lock "$Paid$".

The $Paid$ lock, and locks in general, are used solely to specify when information may flow from storage locations to actors. The 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.

In the case of the program we would need to associate the opening of the $Paid$ lock with the actual confirmation of payment in the code.

The idea is that security policies are associated with the storage locations in a program. In the case of a software key, the policy would then be written as

$$\{vendor; Paid \Rightarrow customer\}$$

The data contained in a storage location with this policy may flow freely to the vendor, but should not flow to the customer until the paid lock is open.

If at some later point the lock was closed again, perhaps because the customer's access to the software key was only for a limited period of time, the data should no longer be accessible to the customer, though they would not be required to forget what they have already learnt.

Note that flow locks is an information flow policy *specification* mechanism – it allows the programmer to specify a flow policy in a program, and get guarantees that the program correctly conforms to the policy as stated. Flow locks makes no attempts to address the issue of whether the policy itself is correctly stated, i.e., that the opening and closing of locks is done in the right places, and that data is labelled with the proper policies. This is largely an orthogonal problem handled by external analyses and verification mechanisms.

Flow Lock Encodings One aim of the flow locks approach is to provide a general language into which a variety of information flow mechanisms can be encoded. As a simple example of such an encoding from [BS06a], consider a system

with data marked with "high" or "low", and a single statement $\ell := declassify(h)$ 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 $\{high; low\}$ (readable by actors $high$ and low). High variables would then be given the policy $\{high; Decl \Rightarrow low\}$, and the statement $\ell := declassify(h)$ can be encoded with the sequence of statements

$$\texttt{open}\ Decl; \ell := h; \texttt{close}\ Decl$$

Let us consider one further example of a policy relating to dynamic change of an information flow lattice. Consider Figure 1 which depicts three information-flow lattices.

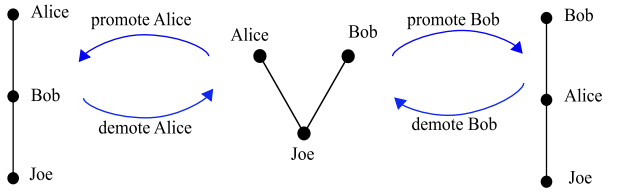

Figure 1. Example Dynamic Policy

In the leftmost lattice Alice is the top element. While Alice is "boss" all information may flow to her. If she is demoted, however, then the information flow lattice changes to the central figure. From there either Bob or Alice can be promoted to be the boss. Let us consider how to encode this intended scenario with flow locks. A semantics for this policy is then provided by the flow locks semantics presented in the remainder of the paper.

To represent this dynamic flow policy with flow locks we begin, not surprisingly, by assuming three actors: *Alice*, *Bob*, and *Joe*. To model the transitions between policies we use two locks: *promoteA* and *promoteB*. The events of promotion and demotion are modelled by the respective opening and closing of these locks. When *promoteA* is open then Alice is boss. Closing *promoteA* (respectively, *promoteB*) corresponds to demoting Alice (resp. Bob).

To complete the picture we need to describe the corresponding policies for the data to be associated with Alice, Bob, and Joe. Joe is the simplest case, and his data has policy $\{Joe; Alice; Bob\}$ – i.e. it is readable by everyone at all times. Alice's data has policy $\{Alice; promoteB \Rightarrow Bob\}$ and Bob has the symmetric policy $\{promoteA \Rightarrow Alice; Bob\}$. For Bob this means that his data is readable by Alice only when Alice has been promoted. Note that if both locks are open then we have a situation not modelled in the figure: Alice and Bob become equivalent from an information flow perspective. If we want to rule this out we cannot do so using the policy on data. We must enforce this via an invariant property of the locks themselves.

Problems with the Flow Lock Security In our previous work we introduced a definition of what it means for a program to be secure with respect to a given assignment of flow-lock policies to variables. Here we highlight some additional issues with this semantics. We will not introduce the definition in its full and gory detail, referring instead to [BS06b] which provides a lengthy stepwise development of the definition.

In common with many of the approaches cited above, flow lock security is defined using a resetting approach based on a flow insensitive version of noninterference called *strong security* introduced by Sabelfeld and Sands [SS00]. This is based on the idea of a bisimulation between any two runs which differ only on the initial values of secrets. Programs P and Q are bisimilar if whenever they are given attacker-equivalent memory states the next computation step of P will be matched by Q and result in low-equivalent states, and the resulting programs will also be bisimilar. This embodies an "aggressive" resetting in that it quantifies over all pairs of low equivalent memories at each step of the bisimulation.

A number of complications in the flow-lock semantics are due to the underlying flow-insensitivity. In particular the definition built in two concepts: *future sensitivity* and *past awareness*. The notion of future sensitivity required that at each step of the bisimulation the security condition had to ensure that any changes to memories would be safe in *all possible* future lock states, even though there is always one specific lock state that would be sufficient to discover such problems. While future sensitivity catches "bad flows" that might happen in the future, *past awareness* deals with permitting "good flows" from the past even though they might *appear* bad at some point in the future. The past-awareness problem forced us to adopt a non-standard semantics where data from control-flow branch points had to be remembered, so that certain programs were not incorrectly flagged as insecure. We will not go into further details of the previous definition here, referring instead to [BS06b]. By recovering flow-sensitivity, our revised definition will simplify the notion of future sensitivity and eliminate the need for past awareness altogether.

A Knowledge-based Approach One of the fundamental difficulties in the bisimulation-style definition is that it builds on a comparison between two runs of a system. While this is fairly intuitive for standard noninterference, 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 to use a more explicit attacker model whereby one reasons about what an attacker learns about the initial inputs to a system as computation progresses [AS07]. The formulation we use here will be closest to [AHSS08].

The basic idea builds on a notion of noninterference described by [DEG06] and 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. Now 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

$$K_2 = \{N \mid \text{ Running } P \text{ on } N \text{ can yield trace } t \text{ followed by } u\ \}$$

We will call K_1 and K_2 *knowledge sets*, and order knowledge sets by $K \sqsubseteq K' \iff K' \subseteq K$. Note that in the above $K_1 \sqsubseteq K_2$: the attackers knowledge increases as the computation proceeds. However, for the program to be considered noninterfering, in all such cases we must have $K_1 = K_2$, since we require the knowledge to not increase at all throughout the program execution.

This style of definition is the key to our new flow lock semantics. The core idea will be 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.

3. Flow Lock Security

In this section we motivate our flow sensitive definition of flow-lock security. The definition is phrased in terms of a labelled transition system where labels represent observable events. We assume an imperative computation model involving commands and stores (memories), but the definition is otherwise not specific to a particular programming language.

3.1 Preliminaries

We begin by recalling the precise language of policies and introduce the base assumptions about the operational semantics of the language.

Policies 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 which we name the *guard* of the clause, and interpret it as a conjunction. Thus for the guard to be satisfied, all the locks $\sigma \in \Sigma$ must be open.

In concrete examples we will often simplify the notation, so that for example we will write (as we did in the introduction) $\{vendor; Paid \Rightarrow customer\}$ instead of

$$\{\varnothing \Rightarrow vendor; \{Paid\} \Rightarrow customer\}\,.$$

A policy p is *less restrictive* than a policy q, written $p \sqsubseteq q$, if for every clause $\Sigma \Rightarrow \alpha$ in q there is a clause $\Sigma' \Rightarrow \alpha$ in p where $\Sigma' \subseteq \Sigma$. For example, $\{vendor; customer\}$ is less restrictive than $\{vendor; Paid \Rightarrow customer\}$ which in turn is less restrictive than $\{vendor\}$. We use the distinguished value $\bot$ to denote the least restrictive policy, for variables that all actors can see at all times. The opposite is the policy $\top$, which is simply the empty set of clauses, meaning no actor could ever see the data of a variable marked with that policy. To join two policies means combining their respective clauses. We define

$$p_1 \sqcup p_2 \equiv \{\Sigma_1 \cup \Sigma_2 \Rightarrow \alpha | \Sigma_1 \Rightarrow \alpha \in p_1,\ \Sigma_2 \Rightarrow \alpha \in p_2\}$$

It should be intuitively clear that the join of two policies is at least as restrictive as each of the two operands, i.e. $p \sqsubseteq p \sqcup p'$ for all p, p'. In contrast, forming the union of two policies, i.e. the meet, corresponding to $\sqcap$, makes the result less restrictive, so we have $p \sqcap p' \sqsubseteq p$ for all p, p'. Both $\sqcap$ and $\sqcup$ are clearly commutative and associative.

We also need the concept of a policy specialized (normalized) to a particular lock state, denoted $p(\Sigma)$, meaning the policy that remains if we remove from all guards the locks which are present in Σ. So for example, if p is $\{Paid \Rightarrow customer\}$, then $p(\{Paid\}) = \{customer\}$. Formally, $p(\Sigma) = \{(\Delta \setminus \Sigma) \Rightarrow \alpha \mid \Delta \Rightarrow \alpha \in p\}$.

Operational Semantics To keep our presentation reasonably concrete we will consider imperative computation modelled by a standard small-step operational semantics defined over configurations of the form $\langle \Sigma, c, M\rangle$ where c (c', d etc.) is a command, M is a memory (store) – a finite mapping from variables to values, and Σ is the lock state – the set of locks that are currently open.

We assume that each channel and variable x, y, ... is assigned a fixed policy, where $pol(x)$ denotes the policy of x.

Transitions in the semantics are labelled $\langle \Sigma, c, M\rangle \xrightarrow{\ell} \langle \Delta, d, N\rangle$ where ℓ is either a distinguished *silent* action τ, or an *observable* action of the form $x(v)$, where x is a channel and v is the value observed on that channel. We let w, w' etc range over observable actions, and $\vec{w}$ a vector of such. We assume the existence of commands which change the lock state. The open and close commands used in the concrete earlier work are sufficient, although other lock-state changing commands are possible. We do, however, assume that whenever the lock state changes then there is no output or memory change, i.e. if $\langle \Sigma, c, M\rangle \xrightarrow{\ell} \langle \Delta, d, N\rangle$ $\Sigma \neq \Delta$ then we must have $M = N$ and $\ell = \tau$. Given the labelled transition system we define some auxiliary notions.

DEFINITION 3.1 (Visibility).

- *We say that* x may be visible to α *if* $\Sigma \Rightarrow \alpha \in pol(x)$ *for some* Σ*; otherwise we say that it is* never visible.
- *We say that* x is visible to α at Δ *if* $\Sigma \Rightarrow \alpha \in pol(x)$ *for some* $\Sigma \subseteq \Delta$*; otherwise we say that it is* not visible at Δ.

We extend these definitions to outputs $x(v)$ *in the same way, and we say that the silent output* τ *is never visible.*

3.2 Motivating the security definition

To motivate our definition we will first look at some properties that we expect it to have. First we consider the case of simple declassification from the introduction. Consider the program $\ell := \mathit{declassify}(h); \ell := h$, which would be encoded as as

$$\texttt{open}\ \mathit{Decl}; \ell := h; \texttt{close}\ \mathit{Decl}; \ell := h$$

The intended meaning of closing a lock is *not* that an actor should forget all they learned while the lock was open. Thus we expect this program to be considered secure, since the value of h is already known at the point of the second assignment. In other words, as we argued in section 2, we expect our definition to be flow *sensitive*, as opposed to our previous, bisimulation-based definition. Practically this means that our semantic definition cannot be a purely local stepwise definition, but requires us to inspect all knowledge gained by an attacker up to a certain assignment. Then we must validate that assignment in the context of the attacker having that knowledge.

Another feature to note is that our flow locks system allows fine-grained flows, in which a secret may be leaked in a series of unrelated steps. The following policy and program exhibits this:

$$x : \{\{\mathit{Day}, \mathit{Night}\} \Rightarrow \alpha\} \quad y : \{\mathit{Night} \Rightarrow \alpha\} \quad z : \{\alpha\}$$
$$\texttt{open}\ \mathit{Day};\ y := x\ ;\texttt{close}\ \mathit{Day};\ \texttt{open}\ \mathit{Night};\ z := y$$

Here (and in subsequent examples) we assume each assignment generates an observable action – i.e. each variable is viewed as an output channel. Here the secret contained in x is leaked into z via y. But at the point where the assignment to z is made, the lockstate in effect does not allow a direct flow from x to z since Day is closed. In addition, at the point where the assignment to y is made, y is not visible at the current lockstate. To verify that this program is allowed, we need to validate the flows at each "level" that the secret flows 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 $\mathit{Actors} \times \mathcal{P}(\mathit{Locks})$.

This leads us to our formal attacker model:

DEFINITION 3.2 (Attacker). *An attacker A is a pair of an actor α and a set of locks Δ, formally*

$$A = (\alpha, \Delta) \in \mathit{Actors} \times \mathcal{P}(\mathit{Locks})$$

We refer to the lockstate component of an attacker as his *capability*, and assume that A can observe locations guarded from α only by locks in Δ.

Intuitively we may think of an attacker as an actor who may open the locks Δ at some point in the future, leading to a *future-sensitive* model[3] that enables us to build secure commands by sequential composition from secure commands (see Section 4).

[3] Future sensitivity is the one component of the original bisimulation-based definition that is retained in this new semantics.

We define attacker visibility as a natural extension of actor visibility, by saying that x is visible to $A = (\alpha, \Delta)$ iff x is visible to α at Δ.

For each attacker we then define the *A-observable transition* $\langle \Sigma, c, M\rangle \xrightarrow{w}_A \langle \Delta, d, N\rangle$ by absorbing transitions which are not visible to attacker A.

DEFINITION 3.3 (A-observable transitions). *We can define the transition relation $\xrightarrow{w}_A$ as the least relation satisfying the following rules:*

$$\frac{\langle \Sigma, c, M\rangle \xrightarrow{w} \langle \Delta, d, N\rangle \qquad w \text{ is visible to } A}{\langle \Sigma, c, M\rangle \xrightarrow{w}_A \langle \Delta, d, N\rangle}$$

$$\frac{\langle \Sigma, c, M\rangle \xrightarrow{\ell} \langle \Sigma', c', M'\rangle \qquad \ell \text{ is not visible to } A \quad \langle \Sigma', c', M'\rangle \xrightarrow{w}_A \langle \Delta, d, N\rangle}{\langle \Sigma, c, M\rangle \xrightarrow{w}_A \langle \Delta, d, N\rangle}$$

We now define some useful compound A-transitions. Firstly define $\langle \Sigma, c, M\rangle \Longrightarrow_A \langle \Delta, d, N\rangle$ if there is a sequence of zero or more transitions from $\langle \Sigma, c, M\rangle$ to $\langle \Delta, d, N\rangle$ with labels not visible to A. Now we define the multi-step A-observable transitions $\xRightarrow{\vec{w}}_A$ for some sequence of output labels $\vec{w}$ by equating $\xRightarrow{\varepsilon}_A$ with $\Longrightarrow_A$ (where ε denotes the empty vector), and by inductively defining

$$\frac{\langle \Sigma, c, M\rangle \xRightarrow{\vec{w}}_A \langle \Sigma', c', M'\rangle \xrightarrow{w}_A \langle \Delta, d, N\rangle}{\langle \Sigma, c, M\rangle \xRightarrow{\vec{w}w}_A \langle \Delta, d, N\rangle}$$

We use the notation $\langle \Sigma, c, M\rangle \xRightarrow{\vec{w}}_A$ as a shorthand for $\exists \Delta, d, N.\, \langle \Sigma, c, M\rangle \xRightarrow{\vec{w}}_A \langle \Delta, d, N\rangle$, i.e. when we don't care what the resulting configuration is.

To reason about attacker knowledge we need to be able to focus on the parts of a memory which are visible to a given attacker.

DEFINITION 3.4 (A-low memory, A-equivalence). *Memory L is A-low for some attacker A if $dom(L) = \{x \mid x \text{ is visible to } A\}$. 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.*

We will adopt the convention that M and N will range over total memories (i.e. their domain will be the set of all variables). With this we can formalize the notion of attacker *knowledge* as follows:

DEFINITION 3.5 (Attacker knowledge). *The knowledge gained by an attacker $A = (\alpha, \Delta)$ from observing a sequence of outputs $\vec{w}$ of a program c starting with a A-low memory L written $k_A(\vec{w}, c, L)$, is defined to be the set of all possible starting memories that could have lead to that observation:*

$$k_A(\vec{w}, c, L) = \{M | M \sim_A L, \langle \Sigma, c, M\rangle \xRightarrow{\vec{w}}_A\}$$

3.3 Flow Lock Security

With this attacker model in hand, we can now formalise our security requirement. 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. To formalize this intuition we first, for convenience, introduce the notion of a *run*. A run is just an output trace together with the lockstate in effect at the time of the last output in the sequence.

DEFINITION 3.6 (Runs). *The set of all runs of a command c starting with lock state Σ and with a starting memory whose A-low projection is L, are defined*

$$Run_A(\Sigma, c, L) = \{(\vec{w}w, \Delta) \mid M \sim_A L, \\ \langle \Sigma, c, M\rangle \stackrel{\vec{w}}{\Longrightarrow}_A \langle \Sigma', c', M'\rangle \stackrel{w}{\rightarrow}_A \langle \Delta, d, N\rangle\}$$

We can now define our security requirement in terms of runs as follows:

DEFINITION 3.7 (Σ Flow Lock Security). *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.

At the top level we can define security for a self-contained program, i.e. one that doesn't assume any locks are open before it starts:

DEFINITION 3.8 (Top-level Flow Lock Security). *A program c is said to be flow lock secure, written $FLS(c)$, iff the program is $\varnothing$-flow lock secure, i.e. $FLS(\varnothing, c)$.*

The above definitions are termination *sensitive*, since they require that no knowledge is gained by the simple observation that there is an output at all. Following [AHSS08] we can define a termination *insensitive* version:

DEFINITION 3.9.
(Termination Insensitive Flow Lock Security)
A program c is said to be termination-insensitive Σ-flow lock secure, written $TIFLS(\Sigma, c)$ iff for all attackers $A = (\alpha, \Delta)$, all A-low memories L, and any two runs $(\vec{w}w, \Omega)$ and $(\vec{w}w', \Omega')$ in $Run_A(\Sigma, c, L)$ such that $\Omega \subseteq \Delta$ we have that

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

In this variant we allow some knowledge to be gained by the last step of the output, but no more than simply learning that there *is* an observable output. See [AHSS08] for more details. Note that by symmetry we compare the knowledge sets under both Ω and Ω'.

4. Basic Properties of Flow Lock Security

In this section we look at some basic properties of the definition of flow lock security. We inspect the basic properties of the definition via the *principles of declassification* as stated by Sabelfeld and Sands [SS05], since flow locks are intended to model various forms of declassification (or more generally *re*classification).

Conservativity The conservativity principle states that in the absence of any declassification the security condition should revert to noninterference. As noted in [BS06a], we can model standard information-flow lattices by policies which contain sets of unguarded actors, so that for example in the two-point lattice $Low \leq High$ we would define two actors low and $high$, and then Low data would be modelled by the policy $\{\varnothing \Rightarrow low; \varnothing \Rightarrow high\}$, whereas $High$ would correspond to $\{\varnothing \Rightarrow high\}$. In the presence of such unguarded policies it is straightforward to see that the notion of flow lock security reduces to the knowledge-based definition of noninterference from [AHSS08].

Monotonicity of release This principle states that adding more declassification to a "secure" program should never render it insecure. In the setting of flow locks, "adding more declassification" is naturally interpreted as *opening more locks*. A secure program which is modified to open more locks (but is otherwise unchanged) will still be secure since it is straightforward to see that the more locks are open in the lockstate at any given point in a trace, the weaker the flow lock security requirement at that point.

Formally we can state the principle of monotonicity as follows:

PROPOSITION 4.1 (Monotonicity of flow lock security). *If $FLS(\Sigma, c)$ and $\Sigma' \supseteq \Sigma$ then $FLS(\Sigma', c)$.*

The proof can be found in the appendix.

Semantic consistency This states that the notion of security should be preserved by any semantics-preserving transformations to a program, and this is true for the semantics we define. One such example is dead code elimination. As mentioned in the introduction, lack of flow sensitivity makes

security definitions sensitive to dead code. Here the definition of flow lock security can never be sensitive to dead code since it only quantifies over possible traces of a system – and these, by definition, are insensitive to dead code.

It is worth noting that semantic consistency is relative to a particular semantics; in the concrete example that we consider in the next section we assume a semantics in which the effect of assignments are directly observable (to an appropriate attacker), something which does not hold for the usual operational semantics. This is referred to as a *semantic anomaly* [SS05], and is common to many security definitions which are phrased in terms of sequences of assignments.

Non-occlusion The non-occlusion principle is the most vague. It tries to capture the requirement that one declassification operation should not be able to mask an arbitrary amount of future insecure information flow. In our system we can argue for non-occlusion as follows. In our definition each assignment is considered in isolation, and the presumed knowledge gained from observing an assignment is exact. Therefore any further knowledge gained by observing any future assignment must still be subject to the same constraints (modulo the knowledge gained by the earlier assignment) with respect to the lock state and policies in force at that time. Adding declassifications therefore cannot mask future unintended flows.

Hookup Properties for Sequential Composition In addition to the basic principles, it is useful to study composition principles (sometimes called *hook-up* properties [McC87]): when can we build secure programs from secure components.

Here we briefly consider the most basic composition principle corresponding to sequential composition. Let us suppose that we have a sequential composition operator (either directly or encodable) with the usual semantics (see the next section for example).

The termination sensitive condition has a technical problem that prevents it from composing sequentially: a program which ends in a silent loop is indistinguishable from one which terminates. This difference is revealed by composing the program with one which performs output. Termination insensitive flow lock security would consider the above composition secure, but still suffers from a problem, though for a different class of programs. A program that either silently terminates or produces one last output before termination is considered secure, since the silent termination is for all purposes equivalent to a silent loop. Composing such a program with one that performs an output again reveals the difference, and causes the previous output to be considered insecure.

To obtain secure composition, the concrete semantics used may thus not have silent termination, i.e. all programs must produce a distinguished visible output if and only if they terminate. We say that such programs have *visible termination*. Our security definition from the previous section is agnostic as to whether visible termination is used or not.

The second minor obstacle to secure sequential composition is the lock state component. For this let us introduce Hoare-like triples $\{\Sigma\}c\{\Sigma'\}$, which state that if any computation of c begins with at least locks Σ open, on termination at least locks Σ' will be open.

PROPOSITION 4.2. *The following proof rule is sound, assuming the concrete semantics uses visible termination:*

$$\frac{FLS(\Sigma, c_1) \qquad \{\Sigma\}c_1\{\Sigma'\} \qquad FLS(\Sigma', c_2)}{FLS(\Sigma, c_1; c_2)}$$

5. Applicability: A Sound Type System

In this section we will illustrate our definition of flow lock security to a specific language and type system, and prove that the type system guarantees flow lock security as given by the definition in the previous section. For the sake of brevity we treat just a simple while-language, but in principle we can apply the same approach to the higher-order language and type system in the style of that studied in [BS06a].

5.1 Language

$$\langle n, M\rangle \Downarrow n \qquad \langle x, M\rangle \Downarrow M[x]$$

$$\frac{\langle e_1, M\rangle \Downarrow v_1 \quad \langle e_2, M\rangle \Downarrow v_2}{\langle e_1 \oplus e_2, M\rangle \Downarrow v_1 \oplus v_2}$$

$$\langle \Sigma, \texttt{open}\ \sigma, M\rangle \xrightarrow{\tau} \langle \Sigma \cup \{\sigma\}, \texttt{skip}, M\rangle$$

$$\langle \Sigma, \texttt{close}\ \sigma, M\rangle \xrightarrow{\tau} \langle \Sigma \setminus \{\sigma\}, \texttt{skip}, M\rangle$$

$$\frac{\langle e, M\rangle \Downarrow v}{\langle \Sigma, x := e, M\rangle \xrightarrow{x(v)} \langle \Sigma, \texttt{skip}, M[x \mapsto v]\rangle}$$

$$\frac{\langle e, M\rangle \Downarrow v \qquad v \in \{\texttt{true}, \texttt{false}\}}{\langle \Sigma, \texttt{if}\ e\ \texttt{then}\ c_{\texttt{true}}\ \texttt{else}\ c_{\texttt{false}}, M\rangle \xrightarrow{\tau} \langle \Sigma, c_v, M\rangle}$$

$$\langle \Sigma, \texttt{while}\ (e)\ c, M\rangle \xrightarrow{\tau} \langle \Sigma, \texttt{if}\ e\ \texttt{then}\ c; \texttt{while}\ (e)\ c\ \texttt{else}\ \texttt{skip}, M\rangle$$

$$\frac{\langle \Sigma, c_1, M\rangle \xrightarrow{\ell} \langle \Sigma', c_1', M'\rangle}{\langle \Sigma, c_1; c_2, M\rangle \xrightarrow{\ell} \langle \Sigma', c_1'; c_2, M'\rangle}$$

$$\langle \Sigma, \texttt{skip}; c_2, M\rangle \xrightarrow{\tau} \langle \Sigma, c_2, M\rangle$$

Figure 2. Operational Semantics

The simple while language presented in Figure 2 will serve as the basis of our presentation. The only two non-standard features are statements `open` σ and `close` σ for manipulating the program's lock state. σ here ranges over single locks. The notion of observable action is defined (as

discussed in the previous section) as the action of assigning to a variable.

5.2 Type System

$$\frac{}{\vdash n : \bot} \qquad \frac{}{\vdash x : pol(x)} \qquad \frac{\vdash e_1 : r_1 \quad \vdash e_2 : r_2}{\vdash e_1 \oplus e_2 : r_1 \sqcup r_2}$$

$$\frac{}{\Sigma \vdash \texttt{open}\ \sigma \rightsquigarrow \top, \Sigma \cup \{\sigma\}} \qquad \frac{}{\Sigma \vdash \texttt{close}\ \sigma \rightsquigarrow \top, \Sigma \setminus \{\sigma\}}$$

$$\frac{}{\Sigma \vdash \texttt{skip} \rightsquigarrow \top, \Sigma} \qquad \frac{\vdash e : r \quad r(\Sigma) \sqsubseteq pol(x)}{\Sigma \vdash x := e \rightsquigarrow pol(x), \Sigma}$$

$$\frac{\vdash e : r \quad \Sigma \vdash c_i \rightsquigarrow w_i, \Sigma_i \quad r \sqsubseteq w_1 \sqcap w_2}{\Sigma \vdash \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \rightsquigarrow w_1 \sqcap w_2, \Sigma_1 \cap \Sigma_2}$$

$$\frac{\vdash e : r \quad \Sigma \cap \Sigma' \vdash c \rightsquigarrow w, \Sigma' \quad r \sqsubseteq w}{\Sigma \vdash \texttt{while}\ (e)\ c \rightsquigarrow w, \Sigma' \cap \Sigma}$$

$$\frac{\Sigma \vdash c_1 \rightsquigarrow w_1, \Sigma_1 \quad \Sigma_1 \vdash c_2 \rightsquigarrow w_2, \Sigma_2}{\Sigma \vdash c_1; c_2 \rightsquigarrow w_1 \sqcap w_2, \Sigma_2}$$

$$\frac{\Sigma \vdash c \rightsquigarrow w, \Sigma'}{\Sigma \vdash c} \quad \text{(Top level judgement)}$$

Figure 3. Flow Lock Type System

The type system we use can be found in figure 3. To simplify the presentation we use only int as base type for expressions, and commands have no base type, so we can restrict ourselves to only the flow locks aspects of the system. However, for convenience we use the boolean values `false` and `true` as shorthands for the value 0, and any value $v \neq 0$, respectively.

For expressions we have judgements of the form $\vdash e : p$ where the policy p, called the *read effect*, is the join of the policies on all variables whose contents are used to produce its result.

For commands the main judgements have the form $\Sigma \vdash c \rightsquigarrow p, \Sigma'$. Here Σ is an assumption about what locks will be open before execution of c. The policy p is the so called *write effect* of a command, which is the union of the policies on all variables whose contents might be changed when executing the command. This plays a similar role to the "PC" level in many information flow type systems. The final component Σ' is a safe approximation (i.e. an underestimation) of the locks that will be open after execution of c.

Since the rules typically mention a number of different policies, we use r and w to range over read effect and write effect policies respectively, to simplify the presentation.

Looking more closely at some of the rules, we note that, unsurprisingly, `open` and `close` are the only commands directly affecting the lock state. In the rule for assignments, the check $r(\Sigma) \sqsubseteq pol(x)$ ensures that the assignment is valid under the current lock state Σ, thereby ruling out leaks from direct flows. In the rule for `if`, the check $r \sqsubseteq w_1 \sqcap w_2$ ensures that no indirect flows leak information about a "secret" conditional expression to "public" locations. Similarly for the test $r \sqsubseteq w$ in the `while` rule.

For the `if` rule, to compute a safe approximation to the locks that will be open it suffices to take the intersection of the resulting lock states of the branches. The `while` rule needs to use a fix point for the resulting lock state since one iteration of the loop may close locks that would then not be open in subsequent iterations.

We note that there is a natural subtyping in the lock state component of this type system. Formally

$$\Sigma \vdash c \rightsquigarrow w, \Sigma' \wedge \Delta \supseteq \Sigma \Longrightarrow \Delta \vdash c \rightsquigarrow w, \Delta' \wedge \Delta' \supseteq \Sigma'$$

This is easily proved by looking at all the uses of the lock states in the rules, and in particular noting that Σ is covariant in $r(\Sigma) \sqsubseteq pol(x)$ in the rule for assignment.

Further, we can formalize our claim that the resulting lock state in the type system is a safe approximation of running the command as follows:

PROPOSITION 5.1 (Lockstate Safety). *$\Sigma \vdash c \rightsquigarrow w, \Sigma'$ implies $\{\Sigma\}c\{\Sigma'\}$.*

The proof of this is a straightforward induction over the typing derivations. We of course also want to prove soundness with respect to progress and preservation. We have that

PROPOSITION 5.2 (Progress). *If $\Sigma \vdash c \rightsquigarrow w, \Delta$ and $c \neq$ `skip` then $\langle \Sigma, c, M \rangle \xrightarrow{\ell} \langle \Sigma', c', M' \rangle$.*

PROPOSITION 5.3 (Preservation). *If $\Sigma \vdash c \rightsquigarrow w, \Delta$ and $\langle \Sigma, c, M \rangle \xrightarrow{\ell} \langle \Sigma', c', M' \rangle$ then $\Sigma' \vdash c' \rightsquigarrow w', \Delta'$ for some $w' \sqsupseteq w$ and $\Delta' \supseteq \Delta$.*

Proof of Progress is a straightforward case on the syntax of commands. Proof of Preservation is much more involved and can be found in the appendix.

Finally we can state the main proposition of this section, which is that the type system implies flow lock security. The type system is formulated in a termination insensitive way, in particular we allow low assignments after high loops, so that is the formulation we will prove.

THEOREM 5.1 (Well typed programs are flow lock secure). *If $\Sigma \vdash c$ then $TIFLS(\Sigma, c)$.*

The proof is given in the appendix.

6. Example Encodings

Many declassification ideas can now be encoded using flow locks – see [BS06b] for some examples. By such an encoding we now obtain a weaker flow-sensitive semantics for the corresponding declassification mechanism.

6.1 Delimited Non-Disclosure

As a simple example let us take a more recent declassification mechanism, *delimited non-disclosure* [BCR08]. In its

simplest form we have variables of either $High$ or Low security levels, and a local block-structured declassification command `declassify` h `in` c which allows a local weakening of the policy so that h is treated as low for the computation of command c. This is a variable-centric variant of Almeida Matos and Boudol's nondisclosure construct [AB05].

To encode this idea using flow locks we need to use one lock $Decl_h$ per high variable h. Then we assign the policy $pol(\ell) = \{high, low\}$ for each low variable ℓ, and $pol(h) = \{high, Decl_h \Rightarrow low\}$ for each high variable h. The encoding of `declassify` h `in` c is then the obvious

$$\texttt{open}\ Decl_h; c; \texttt{close}\ Decl_h$$

For this encoding we need to assume that there are no nested declassifications over the same variable. This is not a real restriction since the inner declassification would be redundant in that case.

The semantics of delimited non-disclosure is bisimulation-based with memory resetting, so suffers from flow insensitivity (see the example in Section 2). We conjecture that our encoding gives a strictly weaker semantics, but that encoded programs typable in our simple type system are also typable in the system given in [BCR08]. This is because our type system is too simple to take advantage of flow sensitivity.

6.2 Gradual Release

A more interesting example is provided by the *Gradual Release* property from [AS07]. This is interesting because the style of definition used there was the inspiration for our approach. Surprisingly we are able to show that, when specialised to the case of simple declassification, our definition coincides exactly with gradual release.

We will begin by presenting their core operational semantics, as well as the Gradual Release security requirement for programs. We will then present a simple encoding of their language using flow locks, and show that for the class of flow locks programs conforming to the encoding, the two operational semantics and security requirements are equivalent. We will also show that their type system is equivalent to the type system given for the example language in Section 5, for that same class of programs.

The language used in [AS07] is a simple while language similar to the one presented in section 5. It uses a simple two-level lattice $\mathcal{L} = \{\text{Low}, \text{High}\}$ with Low $\sqsubseteq$ High and High $\not\sqsubseteq$ Low. As expected data may flow freely from locations marked with Low to locations marked with High, but not the other way around. The special `declassify` command allows a program to leak data from High to Low.

The relevant parts of the operational semantics for this language can be found in Figure 4. There should be no surprises apart from the outputs arising from assignments and declassifications. These are labelled differently – normal assignments to variables marked with Low cause outputs of the form $x(v)$ whereas declassifications output so called

$$\frac{\langle M, e\rangle \Downarrow n \quad pol(x) \neq \text{Low}}{\langle M, x := e\rangle \rightarrow \langle M[x \mapsto v], \texttt{skip}\rangle}$$

$$\frac{\langle M, e\rangle \Downarrow n \quad pol(x) = \text{Low}}{\langle M, x := e\rangle \overset{x(v)}{\rightarrow} \langle M[x \mapsto v], \texttt{skip}\rangle}$$

$$\frac{\langle M, e\rangle \Downarrow n}{\langle M, x := \texttt{declassify}(e)\rangle \overset{r:x(v)}{\rightarrow} \langle M[x \mapsto v], \texttt{skip}\rangle}$$

Figure 4. Operational semantics for Gradual Release

release events denoted by $r : x(v)$. Assignments to variables marked with High do not yield any outputs at all.

The set of all possible low event sequences of a program is defined as follows:

DEFINITION 6.1 (Low event sequences). *The set of all possible low event sequences that program c may generate starting from a low memory L is*

$$GRRun(c, L) = \{\vec{u} \mid M =_{Low} L, \langle M, c\rangle \overset{\vec{u}}{\Longrightarrow} \langle M', c'\rangle\}$$

where $=_{Low}$ is equivalence on the low part of the memory.

For a program to satisfy Gradual Release, it needs to fulfill the following property[4]:

DEFINITION 6.2 (Gradual Release). *A command c satisfies* Gradual Release*, written $GR(c)$, if for all low projections of memories L, and all pairs of sequences $\vec{u}u, \vec{u}u' \in GRRun(c, L)$, we have*

$$k(c, L, \vec{u}u) = k(c, L, \vec{u}u')$$

Flow Locks Encoding The language displayed here is as noted already very similar to that shown in Section 5 and the encoding is straightforward as previously described. We define an encoding function $\widehat{\cdot}$ over commands, and policies etc.

First we need to represent the security levels High and Low. As before we introduce two actors: low is only allowed to see public (Low) data, while $high$ is allowed to see any data. We also introduce a lock $Decl$ to handle declassification. We can then encode the two levels as $\widehat{\text{High}} = \{high; Decl \Rightarrow low\}$ and $\widehat{\text{Low}} = \{high; low\}$. We have $\widehat{\text{Low}} \sqsubseteq \widehat{\text{High}}$ and $\widehat{\text{High}} \not\sqsubseteq \widehat{\text{Low}}$, as expected. We extend the encoding to variables ($\widehat{x}$) and memories ($\widehat{M}$) in the obvious way, by encoding all policies involved.

Secondly, we need to encode declassification. As previously the command

$$x := \texttt{declassify}(e)$$

[4] We take the liberty of presenting the definitions from [AS07] in a style that more closely resembles those which we have used for our own definitions. Our presentation is not different in any substantial way.

is represented with the sequence of commands

$$\texttt{open}\ Decl; x := e; \texttt{close}\ Decl$$

And that is all we need.

Equivalence Our main goal here is to show that our encoding of the Gradual Release primitives leads to a system that is equivalent to the original Gradual Release system presented in [AS07]. In particular, we want to show that a program will be deemed secure according to Gradual Release if and only if its encoding is deemed flow lock secure.

$$GR(c) \equiv TIFLS(\varnothing, \widehat{c})$$

To do this, we first note that on the flow locks side the only possible lockstates at any point in the program are $\mathcal{P}(Locks) = \{\varnothing, \{Decl\}\}$, and the only actors are $Actors = \{high, low\}$. Further we note that for all attackers $A \in Actors \times \mathcal{P}(Locks)$, the only attacker that would not have perfect knowledge of the memory at all times is $A = (low, \varnothing)$. We can then specialise the definition of flow lock security, to say that an encoded program $\widehat{c}$ is termination insensitive flow lock secure iff for attacker $A = (low, \varnothing)$, for all A-low memories $\widehat{L}$, and all pairs of runs $(\vec{w}w, \varnothing), (\vec{w}w', \Omega) \in Run_A(\varnothing, \widehat{c}, \widehat{L})$ we have that

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

Next we note that we have a simple correspondance between the definitions of runs.

LEMMA 6.1 (Correspondence between runs). *If* $(\vec{w}u) \in GRRun(c, L)$ *then* $(\vec{w}w, \Omega) \in Run_A(\varnothing, \widehat{c}, \widehat{L})$ *for* $A = (low, \varnothing)$*, and further* $\Omega = \{Decl\}$ *iff* u *is a release event, otherwise* $\Omega = \varnothing$*.*

The proof of this is a straighforward inspection of c.

Applying lemma 6.1 to our specialized version of flow lock security above, we end up with exactly definition 6.2, which is what we wanted to prove.

Type system equivalence We can also show that the type system presented in [AS07] is equivalent to the type system presented in Section 5. The typing judgements and rules for expressions in the gradual release system are identical to those in our type system. For commands, we have that

$$\vdash_{\text{GR}} c \rightsquigarrow w \iff \varnothing \vdash \widehat{c} \rightsquigarrow \widehat{w}, \varnothing$$

This is trivial to show for all commands except for assignments and declassifications.

For assignments the rule for the Gradual Release system states that

$$\frac{\vdash_{\text{GR}} e : r \quad r \sqsubseteq pol(x)}{\vdash_{\text{GR}} x := e \rightsquigarrow pol(x)}$$

and we have a direct correspondence with the type rule for assignments from Section 5, specialised to encoded commands:

$$\frac{\vdash e : \widehat{r} \quad \widehat{r} \sqsubseteq \widehat{pol(x)}}{\varnothing \vdash x := e \rightsquigarrow \widehat{pol(x)}, \varnothing}$$

For declassification the rule from Gradual Release is simply

$$\frac{\vdash_{\text{GR}} e : r}{\vdash_{\text{GR}} x := \texttt{declassify}(e) \rightsquigarrow pol(x)}$$

i.e. no constraints on the respective security labels of e and x. For the encoded equivalent,

$$\texttt{open}\ Decl; x := e; \texttt{close}\ Decl$$

we can simply construct the derivation and everything is trivially typeable, with the exception of the constraint $r(\{Decl\}) \sqsubseteq pol(x)$ arising from the assignment in the middle. Since we know from the domain that r is either $\{high; low\}$ or $\{high; Decl \Rightarrow low\}$, we have that $r(\{Decl\}) = \{high; low\} = \widehat{\text{Low}}$. Since for all l, $\widehat{\text{Low}} \sqsubseteq l$, the constraint $\widehat{\text{Low}} \sqsubseteq pol(x)$ is always fulfilled, and we are done.

Discussion What we hope to show with this encoding is that this could have been a feasible (not to say easy) way to prove properties about Gradual Release. The proofs here that Gradual Release is a specialisation of Flow Locks are much less involved than the proofs in the Gradual Release paper, even though those are quite simple to begin with.

Gradual Release is a special case in that it is already flow sensitive, so we get an exact equivalence between the original semantics and the flow locks induced one. We could not get such a correspondence with a flow insensitive system. However, we argue that most other systems are not inherently flow insensitive, and that giving a flow sensitive semantics to them via a flow locks encoding is not only feasible, but also beneficial since it makes it easier to relate various semantics and enforcement mechanisms.

Reasoning about flow locks is greatly simplified by the new form of semantics. But what we have not done in these examples is take advantage of the fact that the semantic condition is not only simpler but also more liberal: in fact the type system we have presented is very similar to that which we previously verified against a flow *insensitive* semantics. Flow sensitivity would be useful in cases where the type system also needs to track properties of values – for example if we wanted to extend the typings to additionally verify that openings of locks only occurred in specific states, or released specific parts of some data (c.f. [BNR08]). Any resetting-style semantics would not be able to track such properties through a computation.

7. Related Work

As mentioned previously, the knowledge based approach used in this paper is inspired by the Gradual Release work [AS07]. Similar uses of knowledge sets appear earlier – e.g. [DEG06] – and many of the classic noninterference definitions have a knowledge or "deducability" flavour. However [AS07] appears to be the first to use this style of definition to reason about the semantics of declassification. Gradual release has also been extended by [BNR08] in a rather orthogonal direction, by allowing declassifications to carry a

logical specification of *what* is declassified, and under what condition.

The notion of flow (in)sensitivity comes from the static analysis world, where it is used to characterise program *analyses*, and is not used to describe the underlying semantic property.

The flow insensitivity problem arising in the papers mentioned in the introduction [MS04, EP05, EP03, AB05, Dam06, MR07, BCR08, LM08] all come about through somewhat related bisimulation-like definitions. But flow insensitivity can arise, in varying degrees, in other styles of model too. For example, [SHTZ06] deals with a detailed model of information flow policy updating. The semantics is phrased in terms of the trace segments in between policy updates, and asserts noninterference for the programs at the beginning of each of these segments. This is a resetting approach since it reasserts noninterference at intermediate program points, and thus becomes flow insensitive. As another example, flow insensitivity also arises in the definition of qualified robust declassification from [MSZ04] which uses a "scrambling" semantics for endorsement (upgrading of integrity) which nondeterministically resets the value of a variable after its endorsement.

Flow locks do not deal directly with the question of *what* information is released (e.g. the length of a cryptographic key or the first four digits of a credit card number), and that is one natural direction for further work. In general policy mechanisms that deal with *what* is released are more extensional and therefore less prone to problems of context sensitivity. However a knowledge-style semantics can be useful in that setting too: a recent approach to expressive policies by Banerjee *et al* [BNR08] uses a generalisation of gradual release which is able to express policies which describe what is released (relative to the current state) at specific release points in the code. The policies include regular program assertions and can thus also constrain the conditions of release[5].

Acknowledgements Thanks to the ProSec group at Chalmers and the anonymous referees for useful comments and suggestions. This work was supported by the Swedish research agencies SSF, Vinnova, VR and by the Information Society Technologies programme of the European Commission under the IST-2005-015905 MOBIUS project.

[5] In [BNR08] it is claimed in passing (p351) that the flow locks approach "is subsumed by our approach." It is somewhat unclear in what sense flow locks are subsumed, but presumably this refers to the fact that flow locks can be viewed as boolean shadow variables, and as such can be used to express a conditional release policy in their setting. It not obvious to us how [BNR08] subsumes flow lock policies in any general sense since their work (i) only deals with a two-level low-high lattice, so cannot directly model multilevel security, integrity, and their combination (ii) only deals with declassification, and not e.g. upgrading of data or revocation of a principal's clearance level, (iii) only allows declassification at an atomic step, and (iv) only deals with a termination sensitive security condition.

References

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

[AHSS08] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination insensitive noninterference leaks more than just a bit. In *Proc. European Symp. on Research in Computer Security*, 2008.

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

[BCR08] Gilles Barthe, Salvador Cavadini, and Tamara Rezk. Tractable enforcement of declassification policies. In *Proc. IEEE Computer Security Foundations Symposium*, 2008.

[BNR08] A. Banerjee, D. Naumann, and S. Rosenberg. Expressive declassification policies and modular static enforcement. *IEEE Symposium on Security and Privacy*, pages 339–353, 2008.

[BS06a] N. Broberg and D. Sands. Flow locks: Towards a core calculus for dynamic flow policies. In *Programming Languages and Systems. 15th European Symposium on Programming, ESOP 2006*, volume 3924 of *LNCS*. Springer Verlag, 2006.

[BS06b] N. Broberg and D. Sands. 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] M. Dam. Decidability and proof systems for language-based noninterference relations. In *Proc. ACM Symp. on Principles of Programming Languages*, 2006.

[DEG06] C. Dima, C. Enea, and R. Gramatovici. Nondeterministic nointerference and deducible information flow. Technical Report 2006-01, University of Paris 12, LACL, 2006.

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

[EP05] R. Echahed and F. Prost. Security policy in a declarative style. In *Proceedings of the 7^{th} International Conference on Principles and Practice of Declarative Programming (PPDP '05)*, Lisboa, Portugal, July 2005.

[GM82] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20, April 1982.

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

[McC87] D. McCullough. Specifications for multi-level security and hook-up property. In *Proc. IEEE Symp. on Security and Privacy*, pages 161–166, April 1987.

[MR07] H. Mantel and A. Reinhard. Controlling the what and where of declassification in language-based security. In *Proc. European Symp. on Programming*, volume 4421 of *LNCS*, pages 141–156. Springer-Verlag, March 2007.

[MS04] H. Mantel and D. Sands. Controlled downgrading based on intransitive (non)interference. In *Proc. Asian Symp. on Programming Languages and Systems*, volume 3302 of *LNCS*, pages 129–145. Springer-Verlag, November 2004.

[MSZ04] A. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification. In *Proc. IEEE Computer Security Foundations Workshop*, pages 172–186, June 2004.

[SHTZ06] N. Swamy, M. Hicks, S. Tse, and S. Zdancewic. Managing policy updates in security-typed languages. *Computer Security Foundations Workshop, IEEE*, 2006.

[SS00] A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Proc. IEEE Computer Security Foundations Workshop*, pages 200–214, July 2000.

[SS05] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *Proc. IEEE Computer Security Foundations Workshop*, pages 255–269, June 2005.

[TZ04] Stephen Tse and Steve Zdancewic. Run-time principals in information-flow type systems. In *In IEEE Symposium on Security and Privacy*, pages 179–193, 2004.

[ZM07] L. Zheng and A. C. Myers. Dynamic security labels and static information flow control. *International Journal of Information Security*, 6, 2007.

Appendix

This appendix is not included in the published proceedings version of the paper. It includes proofs of the main results from the paper.

Proof of Proposition 4.1 We need to show that

$$FLS(\Sigma, c) \wedge \Sigma \subseteq \Sigma' \Longrightarrow FLS(\Sigma', c)$$

Assume $FLS(\Sigma, c)$. That means that for all attackers $A = (\alpha, \Delta)$, and all A-low memories L, we have that if $(\vec{w}w, \Omega) \in Run_A(\Sigma, c, L)$ then $\Omega \subseteq \Delta \Longrightarrow k_A(\vec{w}w, c, L) = k_A(\vec{w}, c, L)$

We make the following observations for using $\Sigma' \supseteq \Sigma$:

Changing the lock state will not affect control flow of a program, which means there will be a direct one-to-one mapping between elements in the two traces, with the same last element of the output sequence.

For each element $(\vec{w}w, \Omega') \in Run_A(\Sigma', c, L)$ with a corresponding $(\vec{w}w, \Omega) \in Run_A(\Sigma, c, L)$, we will have that $\Omega' \supseteq \Omega$. The larger lock state is because adding more locks at the start can never lead to fewer locks open at any subsequent point in the program.

We can then see that using $\Omega' \supseteq \Omega$ in the implication is less restrictive since, $\Omega' \subseteq \Delta$ will hold for fewer attackers.

Proof of Proposition 5.3 We need to show that

$$\Sigma \vdash c \rightsquigarrow w, \Delta \wedge \langle \Sigma, c, M\rangle \xrightarrow{\ell} \langle \Sigma', c', M'\rangle \\ \Longrightarrow \Sigma' \vdash c' \rightsquigarrow w', \Delta' \wedge w' \sqsupseteq w \wedge \Delta' \supseteq \Delta$$

which we do by induction of the height of the typing derivation.

Case $c = x := e$: We know

$$\frac{\vdash e : r \quad r(\Sigma) \sqsubseteq pol(x)}{\Sigma \vdash x := e \rightsquigarrow pol(x), \Sigma}$$

and $\langle \Sigma, x := e, M\rangle \xrightarrow{\ell} \langle \Sigma, \texttt{skip}, M'\rangle$ and can show that $\Sigma \vdash \texttt{skip} \rightsquigarrow \top, \Sigma$ where $\top \sqsupseteq pol(x)$.

Case $c = \texttt{open}\ \sigma$: We must have

$$\Sigma \vdash \texttt{open}\ \sigma \rightsquigarrow \top, \Sigma \cup \{\sigma\}$$

and

$$\langle \Sigma, \texttt{open}\ \sigma, M\rangle \xrightarrow{\ell} \langle \Sigma \cup \{\sigma\}, \texttt{skip}, M\rangle$$

and the conclusion follows trivially.

Case $c = \texttt{close}\ \sigma$: Like the case for $\texttt{open}\ \sigma$.

Case $c = \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2$: We must have

$$\frac{\vdash e : r \quad \Sigma \vdash c_i \rightsquigarrow w_i, \Sigma_i \quad r \sqsubseteq w_1 \sqcap w_2}{\Sigma \vdash \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \rightsquigarrow w_1 \sqcap w_2, \Sigma_1 \cap \Sigma_2}$$

and

$$\langle \Sigma, \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2, M\rangle \xrightarrow{\ell} \langle \Sigma, c_i, M\rangle$$

for some $i \in \{1, 2\}$, and we have that $\Sigma \vdash c_i \rightsquigarrow w_i, \Sigma_i$ and $w_i \sqsupseteq w_1 \sqcap w_2$ and $\Sigma_i \supseteq \Sigma_1 \cap \Sigma_2$.

Case $c = \texttt{while}\ (e)\ c$: We have that

$$\frac{\vdash e : r \quad \Sigma \cap \Sigma' \vdash c \rightsquigarrow w, \Sigma' \quad r \sqsubseteq w}{\Sigma \vdash \texttt{while}\ (e)\ c \rightsquigarrow w, \Sigma'}$$

and

$$\langle \Sigma, \texttt{while}\ (e)\ c, M\rangle \\ \xrightarrow{\ell} \langle \Sigma, \texttt{if}\ e\ \texttt{then}\ c; \texttt{while}\ (e)\ c\ \texttt{else}\ \texttt{skip}, M\rangle$$

We can then construct the following derivation:

$$\frac{\begin{array}{cc}\vdash e : r & \Sigma \vdash \texttt{skip} \rightsquigarrow \top, \Sigma \\ \Sigma \vdash c; \texttt{while}\ (e)\ c \rightsquigarrow w, \Sigma' & r \sqsubseteq w\end{array}}{\Sigma \vdash \texttt{if}\ e\ \texttt{then}\ c; \texttt{while}\ (e)\ c\ \texttt{else}\ \texttt{skip} \rightsquigarrow w, \Sigma' \cap \Sigma}$$

To prove that the sequential composition can indeed be typed we continue with

$$\frac{\Sigma \vdash c \rightsquigarrow w, \Sigma'' \quad \Sigma'' \vdash \texttt{while}\ (e)\ c \rightsquigarrow w, \Sigma'}{\Sigma \vdash c; \texttt{while}\ (e)\ c \rightsquigarrow w, \Sigma'}$$

The first premise in this deriviation holds because of subtyping for lock sets, together with the observation that $\Sigma \supseteq \Sigma \cap \Sigma'$. By the subtyping rule we then also know that $\Sigma'' \supseteq \Sigma'$. To show the second premise we observe that

$$\frac{\vdash e : r \quad \Sigma'' \cap \Sigma' \vdash c \rightsquigarrow w, \Sigma' \quad r \sqsubseteq w}{\Sigma'' \vdash \texttt{while}\ (e)\ c \rightsquigarrow w, \Sigma' \cap \Sigma''}$$

and note that this is an equivalent statement since $\Sigma'' \cap \Sigma' = \Sigma'$ due to $\Sigma'' \supseteq \Sigma'$.

Remains to show that $\Sigma' \vdash c \rightsquigarrow w, \Sigma'$. Since $\Sigma' \supseteq \Sigma' \cap \Sigma$ we can show by subtyping of the original premise that $\Sigma' \vdash c \rightsquigarrow w, \Sigma'''$ where $\Sigma''' \supseteq \Sigma'$. To see that $\Sigma''' = \Sigma'$ we note that by the subtyping rule we have that

$$\Sigma''' \setminus \Sigma' \subseteq \Sigma' \setminus (\Sigma' \cap \Sigma) = \Sigma' \setminus \Sigma$$

and the only way to satisfy that inequation is if $\Sigma''' \setminus \Sigma' = \varnothing$, hence $\Sigma''' \subseteq \Sigma'$ and we are done.

Case $c = c_1; c_2$: We have two cases, either $c_1 = \texttt{skip}$ or $c_1 \neq \texttt{skip}$. In the former case the conclusion follows trivially from the typing derivation and semantic rule, so the interesting case is the latter. We then have that

$$\frac{\Sigma \vdash c_1 \rightsquigarrow w_1, \Sigma_1 \quad \Sigma_1 \vdash c_2 \rightsquigarrow w_2, \Sigma_2}{\Sigma \vdash c_1; c_2 \rightsquigarrow w_1 \sqcap w_2, \Sigma_2}$$

and

$$\langle \Sigma, c_1; c_2, M \rangle \xrightarrow{\ell} \langle \Sigma', c_1'; c_2, M' \rangle$$

where the induction hypothesis gives us that

$$\Sigma' \vdash c_1' \rightsquigarrow w_1', \Sigma_1' \wedge w_1' \sqsupseteq w_1 \wedge \Sigma_1' \supseteq \Sigma_1$$

We then have by subtyping that $\Sigma_1' \vdash c_2 \rightsquigarrow w_2, \Sigma_2'$ where $\Sigma_2' \supseteq \Sigma_2$, and thus we have that

$$\Sigma' \vdash c_1'; c_2 \rightsquigarrow w_1' \sqcap w_2, \Sigma_2' \wedge w_1' \sqcap_2 \sqsupseteq_1 \sqcap_2 \wedge \Sigma_2' \supseteq \Sigma_2$$

and we are done.

DEFINITION .1 (Bounded iteration). *We define a bound on iteration of while-loops as follows:*

$$[\texttt{while}\ (e)\ c]_0 = \texttt{skip}$$

$$[\texttt{while}\ (e)\ c]_k = \texttt{if}\ e\ \texttt{then}\ c; [\texttt{while}\ (e)\ c]_{k-1}\ \texttt{else skip}$$

LEMMA .1 (Consistent run).
If $(\vec{w}, \Delta) \in Run_A(\Sigma, c, M\backslash_A)$ *and*

$$\langle \Sigma, c, M \rangle \Longrightarrow_A \langle \Sigma', c', M' \rangle$$

then $(\vec{w}, \Delta) \in Run_A(\Sigma', c', M'\backslash_A)$
Also, if $(w\vec{w}w', \Delta) \in Run_A(\Sigma, c, M\backslash_A)$ *and*

$$\langle \Sigma, c, M \rangle \xrightarrow{w}_A \langle \Sigma', c', M' \rangle$$

then $(\vec{w}w', \Delta) \in Run_A(\Sigma', c', M'\backslash_A)$

The proof follows directly from the construction of $Run_A(\Sigma, c, M\backslash_A)$. Note also that this extends naturally to the case where we take more than one step and/or produce more than one output along the way.

LEMMA .2 (Context typing). *If* $\Sigma \vdash \mathbb{E}[c] \rightsquigarrow w, \Sigma'$*, then* $\Sigma \vdash c \rightsquigarrow w_c, \Sigma''$ *with* $w \sqsubseteq w_c$.

Proof: Straighforward induction on the typing derivation for $\mathbb{E}[c]$.

LEMMA .3 (Deterministic expression evaluation). *If* $\vdash e : r$ *and r is visible to A and* $\langle e, M \rangle \Downarrow v$ *then* $\forall M' \sim_A M$ *we have that* $\langle e, M' \rangle \Downarrow v$

Proof: By induction on the structure of e.

Case $e = n$: We have $\langle n, M \rangle \Downarrow n$ for all M so the conclusion always holds.

Case $e = x$: We have that $\langle x, M \rangle \Downarrow (M[x])$, and since $pol(x)$ is visible to A and $M' \sim_A M$ we know that $M'[x] = M[x]$.

Case $e = e_1 \oplus e_2$: By the assumption and the type rule for operators we know that $r_1 \sqcup r_2$ is visible to A, which implies that r_i is visible to A, $i \in \{1, 2\}$. We apply the induction hypothesis to the subterms, and combine that with $\oplus$ being deterministic, and we are done.

LEMMA .4 (Silent evaluation). *If* $\Sigma \vdash c \rightsquigarrow w, \Delta$ *and* w *is not visible to A, then running* c *with any starting memory will not produce any A-visible output, and will not change the memory in any way visible to A. Formally,* $\forall M$ *we have either*

$$\langle \Sigma, c, M \rangle \Longrightarrow_A \langle \Sigma', \texttt{skip}, M' \rangle$$

with $M' \sim_A M$*, or* $\langle \Sigma, c, M \rangle \Uparrow_A$

Proof: By induction on the height of the typing derivation of $\Sigma \vdash c$.

Case $c = x := e$: We have

$$\frac{\vdash e : r \quad r(\Sigma) \sqsubseteq pol(x)}{\Sigma \vdash x := e \rightsquigarrow pol(x), \Sigma}$$

Since $pol(x)$ is not visible to A, for the transition

$$\langle \Sigma, x := e, M \rangle \xrightarrow{\ell} \langle \Sigma, skip, M[x \mapsto v] \rangle$$

where $\langle e, M \rangle \Downarrow v$, l is not visible to A, and $M[x \mapsto v] \sim_A M$.

Case $c = \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2$: We have

$$\frac{\vdash e : r \quad \Sigma \vdash c_i \rightsquigarrow w_i, \Sigma_i \quad r \sqsubseteq w_1 \sqcap w_2}{\Sigma \vdash \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \rightsquigarrow w_1 \sqcap w_2, \Sigma_1 \cap \Sigma_2}$$

Neither w_1 nor w_2 are visible to A, so we can take a transition step

$$\langle \Sigma, \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2, M \rangle \xrightarrow{\tau} \langle \Sigma, c_i, M \rangle$$

using either transition rule for conditionals. We apply the induction hypothesis to the resulting term and we are done.

Case $c = \texttt{while}\ (e)\ c'$: We have

$$\frac{\vdash e : r \quad \Sigma \cap \Sigma' \vdash c' \rightsquigarrow w, \Sigma' \quad r \sqsubseteq w}{\Sigma \vdash \texttt{while}\ (e)\ c' \rightsquigarrow w, \Sigma' \cap \Sigma}$$

To prove this case we need a contradiction. Assume that running c will produce a first output visible to A on the kth iteration, i.e. after first performing $k-1$ silent iterations. This means that up to the point of the first output, running c will be equivalent to running a bounded iteration $[\texttt{while}\ (e)\ c']_k$ such that:

$$\langle \Sigma, [\texttt{while}\ (e)\ c']_k, M\rangle \Longrightarrow_A \langle \Delta, c'; \texttt{skip}, M'\rangle$$

with $M' \sim_A M$. We must then have

$$\langle \Delta, c'; \texttt{skip}, M'\rangle \overset{w}{\Longrightarrow}_A \langle \Delta', c''; \texttt{skip}, M''\rangle$$

since the output cannot have come from the `skip`. But by the induction hypothesis and the typing of c we know that running c cannot produce any output visible to A, and we have our contradiction.

Case $c = c_1; c_2$: We apply the induction hypothesis to both subterms and we are done.

The remaining cases for c can never produce any output or change the memory so they are trivial.

LEMMA .5 (Deterministic output).
If $\Sigma \vdash$, $M \sim_A N$, $\langle \Sigma, c, M\rangle \overset{\vec{w}w}{\Longrightarrow}_A \langle \Sigma', c', M'\rangle$ *and* $\langle \Sigma, c, N\rangle \overset{\vec{w}w}{\Longrightarrow}_A \langle \Sigma'', c'', N'\rangle$ *then* $c' = c''$ *and* $M' \sim_A N'$.

Proof: By induction on the length of the transition sequence producing $\vec{w}w$ when running with memory M.

Case $c = \mathbb{E}[x := e]$: We have

$$\frac{\vdash e : r \quad r(\Sigma) \sqsubseteq pol(x)}{\Sigma \vdash x := e \rightsquigarrow pol(x), \Sigma}$$

and we identify two cases:

i) $pol(x)$ is visible to A. We must then have

$$\langle \Sigma, \mathbb{E}[x := e], M\rangle \overset{x(v)}{\rightarrow}_A \langle \Sigma, \mathbb{E}[\texttt{skip}], M[x \mapsto v]\rangle$$

and

$$\langle \Sigma, \mathbb{E}[x := e], N\rangle \overset{x(v)}{\rightarrow}_A \langle \Sigma, \mathbb{E}[\texttt{skip}], N[x \mapsto v]\rangle$$

where we have $M[x \mapsto v] \sim_A N[x \mapsto v]$. If this was the final output then we are done, and that forms our base case for the induction. Otherwise we apply the induction hypothesis to the resulting configurations.

ii) $pol(x)$ is not visible to A. We then get

$$\langle \Sigma, \mathbb{E}[x := e], M\rangle \overset{\ell}{\rightarrow} \langle \Sigma, \mathbb{E}[\texttt{skip}], M[x \mapsto v]\rangle$$

and

$$\langle \Sigma, \mathbb{E}[x := e], N\rangle \overset{\ell'}{\rightarrow} \langle \Sigma, \mathbb{E}[\texttt{skip}], N[x \mapsto v']\rangle$$

where neither l nor l' are visible to A, and $M[x \mapsto v] \sim_A N[x \mapsto v']$. We continue by applying the induction hypothesis to the resulting configurations.

Case $c = \mathbb{E}[\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2]$: We have

$$\frac{\vdash e : r \quad \Sigma \vdash c_i \rightsquigarrow w_i, \Sigma_i \quad r \sqsubseteq w_1 \sqcap w_2}{\Sigma \vdash \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \rightsquigarrow w_1 \sqcap w_2, \Sigma_1 \cap \Sigma_2}$$

and we identify two cases:

i) r is visible to A. Then by the deterministic expression evaluation lemma we have $\langle e, M\rangle \Downarrow v \Longrightarrow \langle e, N\rangle \Downarrow v$. We must have

$$\langle \Sigma, \mathbb{E}[\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2], M\rangle \xrightarrow{\tau} \langle \Sigma, \mathbb{E}[c_i], M\rangle$$

and

$$\langle \Sigma, \mathbb{E}[\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2], N\rangle \xrightarrow{\tau} \langle \Sigma, \mathbb{E}[c_i], N\rangle$$

for the same $i \in \{1, 2\}$. We continue by applying the induction hypothesis to the resulting configurations.

ii) r is not visible to A, which means $w_1 \sqcap w_2$ is not visible to A. Then by the silent evaluation lemma, and the fact that we know the computations cannot silently diverge before producing the output we seek, we must have that

$$\langle \Sigma, \mathbb{E}[\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2], M\rangle \Longrightarrow_A \langle \Sigma', \mathbb{E}[\texttt{skip}], M'\rangle$$

and

$$\langle \Sigma, \mathbb{E}[\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2], N\rangle \Longrightarrow_A \langle \Sigma'', \mathbb{E}[\texttt{skip}], N'\rangle$$

where $M' \sim_A M \sim_A N \sim_A N'$. Since the lockstate cannot interfere with the evaluation or output, we can continue by applying the induction hypothesis to the configurations $\langle \Sigma', \mathbb{E}[\texttt{skip}], M'\rangle$ and $\langle \Sigma', \mathbb{E}[\texttt{skip}], N'\rangle$.

All other cases are trivial since only one transition rule applies, and that transition does not change the memory or produce any output. We simply perform that transition and apply the induction hypothesis to the resulting configurations.

LEMMA .6 (Deterministic silent termination). *If* $\Sigma \vdash c$ *and* $M \sim_A N$ *and*

$$\langle \Sigma, c, M\rangle \Longrightarrow_A \langle \Sigma', \texttt{skip}, M'\rangle$$

then either

$$\langle \Sigma, c, N\rangle \Longrightarrow_A \langle \Sigma'', \texttt{skip}, N'\rangle$$

or $\langle \Sigma, c, N\rangle \Uparrow_A$.

Proof: By induction on the length of the transition sequence leading to termination when running with memory M.

Case $c = \texttt{skip}$: This case is only interesting since it forms the base case for the induction. `skip` trivially converges to `skip` in 0 steps with no output.

Case $c = \mathbb{E}[x := e]$: Since we know the computation is silent we must have that $pol(x)$ is not visible to A. We then have

$$\langle \Sigma, \mathbb{E}[x := e], M\rangle \xrightarrow{\ell} \langle \Sigma, \mathbb{E}[\texttt{skip}], M[x \mapsto v]\rangle$$

and

$$\langle \Sigma, \mathbb{E}[x := e], N\rangle \xrightarrow{\ell'} \langle \Sigma, \mathbb{E}[\texttt{skip}], N[x \mapsto v']\rangle$$

for some v, v'. We have that neither l nor l' are visible to A, and $M[x \mapsto v] \sim_A N[x \mapsto v']$, and we can apply the induction hypothesis to the resulting configurations.

Case $c = \mathbb{E}[\texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2]$: We have

$$\frac{\vdash e : r \quad \Sigma \vdash c_i \rightsquigarrow w_i, \Sigma_i \quad r \sqsubseteq w_1 \sqcap w_2}{\Sigma \vdash \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2 \rightsquigarrow w_1 \sqcap w_2, \Sigma_1 \cap \Sigma_2}$$

and we identify two cases:

i) r is visible to A. Then by the deterministic expression evaluation lemma we have $\langle e, M\rangle \Downarrow v \Longrightarrow \langle e, N\rangle \Downarrow v$. We must have

$$\langle \Sigma, \mathbb{E}[\texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2], M\rangle \xrightarrow{\tau} \langle \Sigma, \mathbb{E}[c_i], M\rangle$$

and

$$\langle \Sigma, \mathbb{E}[\texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2], N\rangle \xrightarrow{\tau} \langle \Sigma, \mathbb{E}[c_i], N\rangle$$

for the same $i \in \{1, 2\}$. We continue by applying the induction hypothesis to the resulting configurations.

ii) r is not visible to A, which means $w_1 \sqcap w_2$ is not visible to A. Then by the silent evaluation lemma we must have that

$$\langle \Sigma, \mathbb{E}[\texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2], M\rangle \Longrightarrow_A \langle \Sigma', \mathbb{E}[\texttt{skip}], M'\rangle$$

and either

$$\langle \Sigma, \mathbb{E}[\texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2], N\rangle \Longrightarrow_A \langle \Sigma'', \mathbb{E}[\texttt{skip}], N'\rangle$$

where $M' \sim_A M \sim_A N \sim_A N'$, or
$\langle \Sigma, \mathbb{E}[\texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2], N\rangle \Uparrow_A$. In the latter case we are done, in the former we apply the induction hypothesis to the resulting configurations.

All other cases are trivial since only one transition rule could apply, and that transition does not change the memory nor produce any output. We simply perform that transition and apply the induction hypothesis to the resulting configurations.

Proof of Theorem 5.1 , repeated here for convenience. What we want to prove is $\Sigma \vdash c \Longrightarrow TIFLS(c)$, which expanded means

$$\forall A = (\alpha, \Delta), L, (\vec{w}w, \Omega), (\vec{w}w', \Omega') \in Run_A(\Sigma, c, L)$$

we have that

$$\Delta \supseteq \Omega \Longrightarrow k_A(c, L, \vec{w}w) = k_A(c, L, \vec{w}w')$$

We prove this by showing that we must have $w = w'$, by induction on the length of the computation leading to $\vec{w}w$. We identify two cases:

i) $\vec{w}$ has length greater than 0. Then by the deterministic output lemma, and the fact that we know both computations will produce more output and so cannot diverge, we must have that for $M \sim_A N$:

$$\langle \Sigma, c, M\rangle \xRightarrow{\vec{w}}_A \langle \Sigma', c', M'\rangle$$

and

$$\langle \Sigma, c, N\rangle \xRightarrow{\vec{w}}_A \langle \Sigma'', c', N'\rangle$$

where $M' \sim_A N'$. By the consistent run lemma, subject reduction and non-interference of lockstates we then know that $\Sigma' \vdash c'$ and $(w, \Omega), (w', \Omega'') \in Run_A(\Sigma', c', L')$, where L' is the common A-low projection of M' and N', and we can apply the induction hypothesis to get $w = w'$.

ii) $\vec{w}$ has length 0. We then proceed to case on c.

Case $c = \mathbb{E}[x := e]$: We have

$$\frac{\vdash e : r \quad r(\Sigma) \sqsubseteq pol(x)}{\Sigma \vdash x := e \rightsquigarrow pol(x), \Sigma}$$

and we identify two cases:

i) $pol(x)$ is not visible to A. Then

$$\langle \Sigma, \mathbb{E}[x := e], M\rangle \xrightarrow{\ell} \langle \Sigma, \mathbb{E}[\texttt{skip}], M[x \mapsto v]\rangle$$

and

$$\langle \Sigma, \mathbb{E}[x := e], N\rangle \xrightarrow{\ell'} \langle \Sigma, \mathbb{E}[\texttt{skip}], N[x \mapsto v']\rangle$$

We have that neither l nor l' are visible to A, and $M[x \mapsto v] \sim_A N[x \mapsto v']$, and by the consistent run lemma we must have $(w, \Omega), (w', \Omega') \in Run_A(\Sigma, \mathbb{E}[\texttt{skip}], L)$ where L is the common A-low projection of the resulting memories. We can apply the induction hypothesis to get $w = w'$.

ii) $pol(x)$ is not visible to A. Then the next transition will generate the visible output, so we must have $\Omega = \Omega' = \Sigma$. Then by $r(\Sigma) \sqsubseteq pol(x)$ and $\Delta \supseteq \Sigma$ we know that r is visible to A. Then by the deterministic expression evaluation lemma we know $\langle e, M\rangle \Downarrow v \Longrightarrow \langle e, N\rangle \Downarrow v$, so we must have

$$\langle \Sigma, \mathbb{E}[x := e], M\rangle \xrightarrow{x(v)}_A \langle \Sigma, \mathbb{E}[\texttt{skip}], M[x \mapsto v]\rangle$$

and

$$\langle \Sigma, \mathbb{E}[x := e], N\rangle \xrightarrow{x(v)}_A \langle \Sigma, \mathbb{E}[\texttt{skip}], N[x \mapsto v]\rangle$$

We have $w = w' = x(v)$ and we are done.

Case $c = \mathbb{E}[\texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2]$: We have

$$\frac{\vdash e : r \quad \Sigma \vdash c_i \rightsquigarrow w_i, \Sigma_i \quad r \sqsubseteq w_1 \sqcap w_2}{\Sigma \vdash \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2 \rightsquigarrow w_1 \sqcap w_2, \Sigma_1 \cap \Sigma_2}$$

and we identify two cases:

i) r is not visible to A. Then by the silent evaluation lemma and $r \sqsubseteq w_1 \sqcap w_2$ we know the subterms cannot produce A-visible output. We must have

$$\langle \Sigma, \mathbb{E}[\texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2], M\rangle \Longrightarrow_A \langle \Sigma', \mathbb{E}[\texttt{skip}], M'\rangle$$

and

$$\langle \Sigma, \mathbb{E}[\texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2], N\rangle \Longrightarrow_A \langle \Sigma'', \mathbb{E}[\texttt{skip}], N'\rangle$$

with $M' \sim_A M \sim_A N \sim_A N'$. By the consistent run lemma we must also have $(w, \Omega), (w', \Omega'') \in Run_A(\Sigma', \mathbb{E}[\texttt{skip}], L)$ and we can apply the induction hypothesis to get $w = w'$.

ii) r is visible to A. Then by the deterministic expression evaluation lemma we know $\langle e, M\rangle \Downarrow v \Longrightarrow \langle e, N\rangle \Downarrow v$ and we must have

$$\langle \Sigma, \mathbb{E}[\texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2], M\rangle \xrightarrow{\tau} \langle \Sigma, \mathbb{E}[c_i], M\rangle$$

and

$$\langle \Sigma, \mathbb{E}[\texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2], N\rangle \xrightarrow{\tau} \langle \Sigma, \mathbb{E}[c_i], N\rangle$$

for some $i \in \{1, 2\}$. By the consistent run lemma we must have $(w, \Omega), (w', \Omega'') \in Run_A(\Sigma, \mathbb{E}[c_i], L)$ and we can apply the induction hypothesis to get $w = w'$.

All other cases are trivial since only one transition rule applies, and that transition does not change the memory or produce any output. We simply perform that transition, note that the consistent run lemma applies, and apply the induction hypothesis to the resulting configurations.