A Monadic Analysis of Information Flow Security with Mutable State

Karl Crary    Aleksey Kliger    Frank Pfenning
Carnegie Mellon University

Abstract

We explore the logical underpinnings of higher-order, security-typed languages with mutable state. Our analysis is based on a logic of information flow derived from lax logic and the monadic metalanguage. Thus, our logic deals with mutation explicitly, with impurity reflected in the types, in contrast to most higher-order security-typed languages, which deal with mutation implicitly via side-effects.

More importantly, we also take a store-oriented view of security, wherein security levels are associated with regions of the mutable store. In contrast, most other accounts are value-oriented, in that security levels are associated with individual values. Our store-oriented viewpoint allows us to address information flow security while still using a largely conventional logic, but we show that it does not lessen the expressive power of the logic. An interesting feature of our analysis lies in its treatment of upcalls (low-security computations that include high-security ones), employing an “informativeness” judgment indicating under what circumstances a type carries useful information.

1 Introduction

Security-typed languages use a type system to track the flow of information within a program to provide properties such as secrecy and integrity. Secrecy states that high-security information does not flow to low-security agents, and integrity dually states that low-security agents cannot corrupt high-security information. In this paper, we will restrict our attention to secrecy properties. A variety of security-typed languages have been proposed, and several of them are both higher-order (i.e., support first-class functions) and provide mutable state [4, 8, 10, 12, 19].

In this paper, we explore the logical underpinnings of higher-order, security-typed languages with mutable state. Although most such languages rely on side-effects to manage mutable state, a logical analysis requires us to make store effects explicit. Thus, an appropriate programming language is Moggi’s monadic metalanguage [6, 7], and the corresponding logic (via a Curry-Howard isomorphism) is lax logic [2].

Our presentation of lax logic is based on that of Pfenning and Davies [9]. The principal distinctive feature of Pfenning and Davies’s account is a syntactic distinction between terms and expressions, where terms are pure and expressions are (possibly) effectful. They show that this distinction allows the logic to possess some desirable properties (local soundness and local completeness) that state in essence that the logic’s presentation is canonical. Although these properties are not particularly important here, the distinction also provides a clean separation between the pure and effectful parts of our analysis, which greatly simplifies our system.

A basic but important novelty of our account lies in the way we structure the security discipline. Most security-typed languages associate security levels with values, thus producing a situation in which some values are better than others. Although one could doubtless build a logic based on this structure (perhaps building on Abadi et al. [1]), it would certainly differ from the conventional logic in which one value is as good as another.

Instead, we adopt a structure in which security is associated with the mutable store and with operations on that store. Not only does this provide a more conventional logic, it also meshes nicely with our focus on effects. A natural question is whether this store-oriented security discipline limits the expressive power of our account relative to ones based on a value-oriented discipline, but we show (in Section 5) that it does not.

Overview The static semantics of our analysis is based on two typing judgments, one for terms (M) and one for expressions (E). Recall that terms are pure and that security is associated with effects, so the typing judgment for terms makes no mention of security levels. Thus, the typing judgment takes the form $\Sigma, \Gamma \vdash M : A$ (where $\Gamma$ is the usual context and $\Sigma$ assigns a type to the store).

Expressions, on the other hand, may have effects and therefore may interact with the security discipline. Each location in the store has a security level associated with it indicating the least security level that is authorized to read that location. Thus, the typing judgment for expressions tracks the security levels of all locations an expression reads or writes. Only the reads are of direct importance to the security discipline (recall that we do not address integrity), but writes must also be tracked since they provide a means

Submitted for publication to POPL 2004.
of information flow. The judgment takes the form:

$$\Sigma, \Gamma \vdash E \downarrow_{(r, w)} A$$

indicating that $$r$$ is a upper bound to the levels of $$E$$’s reads, and $$w$$ is a lower bound to the levels of its writes, and also that $$E$$ has type $$A$$. Naturally we require that $$r \subseteq w$$, or else $$E$$ would manifestly be leaking information.

In lax logic, expressions are internalized as terms using the monadic type $$\bigcirc A$$. A term of type $$\bigcirc A$$ is a suspended expression of type $$A$$. Thus, the introduction form for the monadic type is a term construct, and the elimination form (which releases the suspended expression) is an expression construct. Similarly, our expressions are internalized as terms using a monadic type written $$\bigcirc_{(r, w)} A$$. Since the effects of the suspended expression will be released when the monad is eliminated, the levels of those effects must be recorded in the monad type.

Most of the rules in our account follow from the intuitions above. One remaining novelty deals with the information content of types. Ordinarily, an expression would be deemed to be leaking information if it were to read from a high-security location, use the result of the read to form a value, and pass that value to a low-security computation. However, that expression would not be leaking information if one could show that the type of that value contained no information, or contained information usable only by a high-security computation (who could have performed the read anyway). Thus the type system contains a judgment $$\Sigma \vdash A \not\rightarrow a$$ stating that the type $$A$$ contains information only for computations at the level $$a$$ at least. This notion of informativeness is essential to accounting for the key issue of upcalls (low-security computations that include high-security computations).

The remainder of this paper is organized as follows: In Section 2 we present our basic logical account, including static and dynamic semantics, but omitting the key issue of upcalls. In Section 3 we extend our account to deal with upcalls. In Section 4 we state and prove a non-interference theorem. In Section 5 we show that our store-oriented account counts by giving an embedding of the SLam calculus into our language. Section 6 discusses some related work, Section 7 offers some concluding remarks.

2 The Secure Monadic Calculus

We begin by describing the syntax, evaluation rules and an initial set of typing rules for our language. While this language will be secure, as we will see in the next section its type system rules out too many programs to be practical. By including some additional rules, we will be able to make it more useful while still retaining the required secrecy property.

2.1 Syntax

As in other work on information flow, we have in mind an arbitrary fixed lattice (that is, a partial order $$\langle \mathcal{L}, \sqsubseteq \rangle$$ equipped with a join $$\sqcup$$, meet $$\sqcap$$, and least $$\bot$$ and greatest $$\top$$ elements) of security levels. We use the meta-variables $$a, b, c, r, w, \zeta$$ to range over elements of $$\mathcal{L}$$.

The full syntax of our language is given in Figure 1. The language is split into two syntactic categories: terms $$M$$ and the expressions $$E$$, following Pfenning and Davies [9]. The terms are pure and evaluated to values $$V$$, while the expressions are executed for effect.

Operation levels To track the flow of information, we classify expressions not only by the value that they return, but also by the security levels of their effects. In particular, we keep track of an operation level $$o = (r, w)$$, for each expression. The security level $$r$$ is an upper bound on the security levels of the store locations that the expression reads, while $$w$$ is a lower bound on the security level of the store locations to which it writes.

Since expressions that write at a security level below their read level are obviously insecure, we restrict the operation levels to be elements of the set $$\mathcal{O}$$:

$$\mathcal{O} = \{(r, w) \in \mathcal{L} \times \mathcal{L} \mid r \subseteq w\}$$

Henceforth, when we write an operation level $$(r, w)$$, we will implicitly assume that it is an element of $$\mathcal{O}$$.

The operation levels have a natural ordering $$(r, w) \preceq (r’, w’)$$.

Given some expression $$E$$, if it reads from level at most $$r$$, then it surely reads from level at most $$r’$$, provided that $$r \subseteq r’$$. Similarly, if it writes at level at least $$w$$, then it writes at level at least $$w’$$, provided that $$w’ \subseteq w$$. That is, operation levels are covariant in the reads and contravariant in the writes:

$$(r, w) \preceq (r’, w’) \iff (r \subseteq r’ \text{ and } w’ \subseteq w)$$

There is a subsumption principle for operation levels: if expression $$E$$ has operation level $$o$$, and $$o \preceq o’$$, then $$E$$ has operation level $$o’$$.

Terms At the term level, we have variables, unit, booleans and conditional terms, function abstractions and applications. In support of our operational semantics, store locations are also terms. With each location $$\ell$$ is associated a fixed security level $$\text{Level}(\ell)$$. The store associates locations with the values they contain. A subtyping relation (explored later in this paper), allows us to treat store cells as either read-write, read-only, or write-only.

The term $$\text{val} E$$ allows expressions to be included at the term level as an element of the monadic type $$\bigcirc_{(r, w)} A$$. Since terms are pure, a $$\text{val} E$$ does not execute the expression $$E$$, but rather represents a suspended computation.

Expressions The expressions include a trivial return expression $$\langle M \rangle$$. The return expression has no effect, and simply returns the value to which $$M$$ evaluates. In general, when an expression has no read effects, we say its read level is $$\bot$$, and if an expression has no write effects, we say its write level is $$\top$$. Accordingly, the operation level of $$\langle M \rangle$$ is $$\langle \bot, \top \rangle$$. Note that $$\langle \bot, \top \rangle$$ is the least element in the $$\sqsubseteq$$ ordering, so our subsumption principle will let us weaken the operation level of $$\langle M \rangle$$ to any operation level.

The sequencing expression let $$x = M$$ in $$F$$ evaluates $$M$$ down to some $$\text{val} E$$, and executes $$E$$ followed by $$F$$. The return value of expression $$E$$ is bound to the variable $$x$$ in $$F$$. If $$E$$ and $$F$$ both have operation level $$o$$, then so does the sequencing expression.

We will often write let $$x = E$$ in $$F$$ as syntactic sugar for let $$\text{val} x = \text{val} E$$ in $$F$$.

In addition, there are expressions that allocate, read from, and write to the store. A read expression $$!M$$ has
operation level \((a, \top)\), where \(a\) is the security level of the store location being read, and returns the contents of the store location. Dually, a write expression \(M := N\) has operation level \((\bot, a)\) and updates the store location with the value of \(N\); it does not return an interesting value (i.e., it returns unit).

Store allocation \(\text{ref}_a (M : A)\) specifies the security level \(a\) and type \(A\) of the new store location.

Allocation cannot leak information. Evidently, it is not a read operation. Less obviously, it is not a write operation either. With a write, another expression may learn something about the current computation by observing a change in the value stored at a particular store location. However, the key to this scenario is that the same location is mentioned by more than one expression. On the other hand, allocation creates a new location that is mentioned nowhere else. Thus, there can be no implicit flow of information via an allocation expression. As a result, allocation has operation level \((\bot, \top)\).

**States** A computation state is a partially executed program, and consists of a triple \((H, \Sigma, E)\) of a store \(H\), a store type \(\Sigma\) and a closed expression \(E\). The store maps locations to values, and the store type maps locations to the types of those values.

We assume that in a state \((H, \Sigma, E)\), the store binds occurrences of store locations \(\ell\) in \(H\) and \(E\), and we identify computation states up to renaming of store locations. In addition, as usual, we identify all constructs up to renaming of bound variables.

### 2.2 Static Semantics

The typing rules of our language consists of two main mutually recursive judgments for typing terms and expressions, and some judgments for typechecking stores, and computation states that are summarized in Figure 2. The first judgment

\[
\Sigma; \Gamma \vdash M : A
\]

says that the term \(M\) has type \(A\) in the context \(\Gamma\), where the store has type \(\Sigma\). The second judgment typechecks expressions

\[
\Sigma; \Gamma \vdash E \gamma_o A
\]

says that the expression \(E\) returns a value of type \(A\) and performs only operations within level \(o\), as discussed above. Each rule is given with its rule number, and the full set of rules appears in Appendix B.

We assume that contexts \(\Gamma\) are well-formed, that is, they contain at most one occurrence of each variable \(x\). We tacitly rename bound variables prior to adding them to a context to maintain well-formedness. Similarly, we assume that store types are well-formed, that is, they contain at most one occurrence of each store location \(\ell\).

**Terms** The typing rules for terms are unsurprising for a simply-typed lambda calculus with unit, abstraction and applications. A store location \(\ell\) (provided that it is in \(\text{dom}(\Sigma)\)) has a \(\text{ref}\)-type with its security level:

\[
\Sigma; \Gamma \vdash \ell : \text{ref}_{\text{Level}(\ell)} \Sigma(\ell) \tag{24}
\]
A computation term \( \text{val} \ E \) has the type \( \bigcirc_o A \), provided the expression \( E \) has type \( A \) and operation level \( \alpha \):

\[
\Sigma; \Gamma \vdash E \vdash \alpha \bigcirc A
\]

(27)

Expressions The typing rules for expressions follow our informal description. Trivial computations have the type of their return value, and operation level \( (\bot, \top) \):

\[
\Sigma; \Gamma \vdash M : A \\
\Sigma; \Gamma \vdash \text{let val } x = M \text{ in } E \vdash \alpha \bigcirc A
\]

(29)

The sequencing expression typechecks provided both of the sub-computations have the same operation level (which may require using the weakening rule for operation levels):

\[
\Sigma; \Gamma \vdash \text{let val } x = M \text{ in } E \vdash \alpha \bigcirc A
\]

(30)

Allocation returns a new read/write store location:

\[
\Sigma; \Gamma \vdash M : A \\
\Sigma; \Gamma \vdash \text{ref}_a (M : A) \vdash \alpha (\bot, \top) \text{ref}_a A
\]

(31)

For read and write expressions we only require that the corresponding store location is readable or writable, respectively:

\[
\Sigma; \Gamma \vdash M : \text{ref}_a A \\
\Sigma; \Gamma \vdash \text{ref}_a M : \alpha (\bot, \top) \text{ref}_a A
\]

(32)

\[
\Sigma; \Gamma \vdash M : \text{refw}_a A \\
\Sigma; \Gamma \vdash N : A \\
\Sigma; \Gamma \vdash M := N \vdash \alpha (\bot, \top) 1
\]

(33)

In general we may weaken the operation level of a computation (indeed, as noted above, this is often necessary for the letval typing rule to apply):

\[
\Sigma; \Gamma \vdash E \vdash \alpha \bigcirc A \\
\Sigma; \Gamma \vdash E \vdash \alpha' \bigcirc A
\]

(34)

Subtyping A subsumption rule allows us to weaken the type \( A \) of a term \( M \) or an expression \( E \), provided \( A \) is a subtype of \( B \):

\[
\Sigma; \Gamma \vdash M : A \\
\Sigma; \Gamma \vdash E \vdash \alpha \bigcirc A \\
\Sigma; \Gamma \vdash F \vdash \alpha \bigcirc B
\]

Subtyping

(28)

\[
\Sigma; \Gamma \vdash M : B \\
\Sigma; \Gamma \vdash E \vdash \alpha \bigcirc A \\
\Sigma; \Gamma \vdash F \vdash \alpha \bigcirc B
\]

Subtyping

(36)

Read-only store cells are covariant in the type of their contents and in their security level, and dually write-only cells are contravariant in each:

\[
\vdash A \leq B \\
\vdash \text{ref}_a A \leq \text{ref}_a B
\]

(17)

\[
\vdash B \leq A \\
\vdash \text{ref}_w a A \leq \text{ref}_w w B
\]

(18)

Read/write store cells are neither covariant nor contravariant, but may be weakened to read-only or write-only cells:

\[
\vdash A \leq B \\
\vdash \text{ref}_a A \leq \text{ref}_a B
\]

(15)

\[
\vdash B \leq A \\
\vdash \text{ref}_w a A \leq \text{ref}_w w B
\]

(16)

Finally, the monadic type \( \bigcirc_o A \) is covariant in the return value type and operation level

\[
\vdash A \leq B \\
\vdash \bigcirc_o A \leq \bigcirc_o' B
\]

(14)

Stores and states A store \( H \) is well-typed with store type \( \Sigma \), provided that each value \( V_i \) in the store is well typed under \( \Sigma \) and the empty context, where \( \Sigma \) has the same domain as \( H \)

\[
\text{dom}(\Sigma) = \{ \ell_1, \ldots, \ell_n \} \\
\Sigma; \vdash V_i : \Sigma(\ell_i) \text{ for } 1 \leq i \leq n
\]

(37)

(Note that since \( \Sigma \) appears on the left in the premise of the rule, it must be well-formed).

A computation state \( (H, \Sigma, E) \) is well-typed provided that the store and the expression are each well-typed with the same store type:

\[
\vdash H : \Sigma \\
\Sigma; \vdash E \vdash \alpha \bigcirc A
\]

(38)

2.3 Operational Semantics and Safety

A computation state is called terminal if it is of the form \((H, \Sigma, [V])\). An evaluation relation \( S \rightarrow S' \) gives the small-step operational semantics for computation states. We write \( S \parallel S' \) if for some terminal state \( S' \), \( S \rightarrow^* S' \). Since terms are pure and do not have an effect on the store, their evaluation rules may be given simply by the relation \( M \rightarrow M' \) (no store is required). The entire set of evaluation rules is given in Figure 3.

We write \( M[N/x] \) and \( E[N/x] \) for the capture-avoiding substitution of \( N \) for \( x \) in the term \( M \) or expression \( E \). We write \( H[\ell \mapsto V] \) for finite map that extends \( H \) with \( V \) at \( \ell \).

A computation state \( S \) is called stuck if it is not terminal and there is no \( S' \) such that \( S \rightarrow S' \). A type-safety theorem shows that if \( S \) is well-typed, it does not become stuck. It is proved using the usual Preservation and Progress Lemmas (see the companion technical report [3] for proofs):

Lemma 2.1 (Preservation). If \( \vdash S \vdash \alpha A \) and \( S \rightarrow S' \) then \( \vdash S' \vdash \alpha A \)

Lemma 2.2 (Progress). If \( \vdash S \vdash \alpha A \) then \( S \) is not stuck.

Although we do not formally prove a non-interference theorem at this stage, it should be fairly clear that any well-typed computation state must have the non-interference property. Consider a simple two-element security lattice \( \mathcal{L} = \{ \bot \subseteq \top \} \). Since an expression \(! \ell \) where \( \text{Level}(\ell) = \top \) will have an operation level \((\top, \top)\), and since there is no way to force the read part of the operation level down once it has been increased, any computation that contains a high-security read will be forced to have operation level \((\top, \top)\). As a result any well-typed computation state with operation level \((\bot, \bot)\) or \((\bot, \top)\) cannot depend on the values in high-security store locations.

3 Upcalls

Although the approach discussed so far is secure, it falls short of a practical language. There is no way to include a computation that reads from the high-security store in a larger low-security computation. In any program with a high security read, the read level of the entire program is pushed up. However, many programs that contain upcalls to high security computations followed by low security code are secure.

Consider the program let \( z = P \rightarrow E \) where \( \text{Level}(\ell) = 1 \) and \( E \) has operation level \((\bot, \bot)\). As we argued in the introduction, \( P \) does not leak information because 1 carries no
\[
M \rightarrow M'
\]

\[
\begin{align*}
if M & \text{ then } N_1 \text{ else } N_2 & \rightarrow & M' \text{ then } N_1 \text{ else } N_2 & \text{ If1} \\
if \text{true} & \text{ then } N_1 & \rightarrow & N_1 & \text{ IfTrue} \\
if \text{false} & \text{ then } N_2 & \rightarrow & N_2 & \text{ IfFalse} \\
M & \rightarrow M' & \text{APP1}
\end{align*}
\]

\[
\begin{align*}
MN & \rightarrow M'N & \text{APP2} \\
VN & \rightarrow VN' & \text{APP}
\end{align*}
\]

\[
(\lambda x : A.M)V \rightarrow M[V/x] & \text{APP}
\]

\[
S \rightarrow S'
\]

\[
\begin{align*}
M & \rightarrow M' & \text{RET1} \\
(H, \Sigma, [M]) & \rightarrow (H, \Sigma, [M']) & \text{LETVAL1} \\
M & \rightarrow M' & \text{LETVALVAL} \\
(H, \Sigma, E) & \rightarrow (H', \Sigma', E') & \text{LETVAL} \\
(H, \Sigma, \text{let val } x = M \text{ in } E) & \rightarrow (H, \Sigma, \text{let val } x = M' \text{ in } E') & \text{LETVAL1} \\
\end{align*}
\]

\[
\begin{align*}
M & \rightarrow M' & \text{REF1} \\
(H, \Sigma, \text{ref}_a(M : A)) & \rightarrow (H, \Sigma, \text{ref}_a(M' : A)) & \text{REF} \\
\ell \notin \text{dom}(H) & \text{ Level(\ell) = a} & \text{LEVEL} \\
(H, \Sigma, \text{ref}_a(V : A)) & \rightarrow (H[\ell \mapsto V], \Sigma[\ell : A], [\ell]) & \text{REF1} \\
\end{align*}
\]

\[
\begin{align*}
M & \rightarrow M' & \text{BANG1} \\
(H, \Sigma, !M) & \rightarrow (H, \Sigma, ![M']) & \text{BANG} \\
(H, \Sigma, !M) & \rightarrow (H, \Sigma, [H(!)]) & \text{BANG} \\
M & \rightarrow M' & \text{ASSN1} \\
(H, \Sigma, M := N) & \rightarrow (H, \Sigma, M' := N) & \text{ASSN1} \\
N & \rightarrow N' & \text{ASSN2} \\
(H, \Sigma, V := N) & \rightarrow (H, \Sigma, V := N') & \text{ASSN2} \\
\ell \in \text{dom}(H) & \text{ Level(\ell) = a} & \text{LEVEL} \\
(H, \Sigma, \ell := V) & \rightarrow (H[\ell \mapsto V], \Sigma, [\ast]) & \text{ASSN}
\end{align*}
\]

Figure 3: Operational Semantics
information. Thus we would like to give the entire program the operation level \((\perp, \perp)\). However the type system we have presented so far would instead promote the operation level of \(E\) and the entire program to \((\top, \top)\).

In order to have a logic of information flow, we must offer an account of upcalls. Indeed, the power to perform high security computations interspersed in a larger low-security computation is the sine qua non of useful secure programming languages. We offer a detailed analysis of two cases where upcalls do not violate our intuitive notion of security. From these examples, we develop a general principle for treating upcalls. We take up the question of non-interference in Section 4.

3.1 An example with unit

Let \(E\) be some expression with type \(A\) and operation level \((r, w)\) (recall that this implies that \(r \subseteq w\)). In general, \(E\) may read values from store locations with security level below \(r\), write values to store locations with security level at least \(w\), and return some value of type \(A\).

Suppose that \(A = 1\). In that case, no matter what \(E\) does, if it terminates, it must return \(\ast\). The return value is not informative.\(^3\) Any other computation \(F\) that may gain information through the execution of \(E\) must be able to read store locations at security level at least \(w\). But since \(r \subseteq w\), \(F\) could just directly read any store locations that \(E\) reads.

On the other hand, any computation with operation level \((r', w')\) where \(w \not\subseteq r'\) can neither observe \(E\)’s effects nor gain any information from its (uninformative) return value.

As a result, in either case, we can say that \(E\) has an effective read level of \(\perp\) just as if it had no reads:

\[
\Sigma; \Gamma \vdash E \Rightarrow_{(r, w)} 1 \quad (\ast)
\]

Note that the read level now refers only to informative reads, not all reads.

The new rule allows us to have some high-security computations prior to low security ones. Suppose \(\Sigma; \vdash E \Rightarrow_{(\top, \top)} 1\), and \(\Sigma; \vdash x : 1 \Rightarrow F \Rightarrow_{(\perp, \perp)} A\) for some \(A\). That is, \(E\) is a high-security computation, and \(F\) is a low-security one. With the new rule, the upcall to \(E\), followed by the low-security computation \(F\), can be type checked using the new rule \((\ast)\), \(E\) has operation level \((\perp, \top)\), which can be weakened to \((\perp, \perp)\) by rule (34), and thus:

\[
\Sigma; \vdash E \Rightarrow_{(\perp, \perp)} 1
\]

\[
\Sigma; \vdash \text{val } x = \text{val } E \in F \Rightarrow_{(\perp, \perp)} A
\] (30)

Note that the rule \((\ast)\) does not alter the write level of the expression (that is, the operation level in the conclusion is not \((\perp, \top)\)). Such a rule would allow programs to leak information.

3.2 A more general example

Now consider a computation \(E\) with operation level \((r, w)\), but this time, suppose that \(E\) has type \(\text{ref}_a B\) for some type \(B\). Are there any situations where \(E\) may be given a different operation level?

Suppose that \(r \subseteq a\). In that case, any computation that may read the \(\text{ref}_a B\) is also able to read any store locations that \(E\) may read. Again, any computation can either do what \(E\) does itself, or it cannot gain information from \(E\)’s return value.

On the other hand, consider the case where \(r \not\subseteq a\). In that case, the particular value of type \(\text{ref}_a B\) that \(E\) returns may carry information from store locations at security level \(r\). For example, \(E\) may return one of two such store locations \(\ell_1\) or \(\ell_2\) from level \(a\) based on some boolean value \(V\) from a store location at security level \(r\). In that case, a computation that reads at security level \(a\) may learn something about \(E\)’s reads (at level \(r\)) by reading from \(E\)’s return value. Since \(r \not\subseteq a\), this represents a violation of secure information flow.

So if \(E\) returns a \(\text{ref}_a B\), we can denote its return level whenever \(r \subseteq a\), because any computation that wishes to make use of that return value would need a read level of at least \(r\). In other words, a \(\text{ref}_a B\) is informative only to computations that may read at least at some security level (namely \(a\)) above \(r\).

Thus, we may wish to add a new rule for \(\text{ref}_a B\):

\[
\Sigma; \Gamma \vdash E \Rightarrow_{(r, w)} \text{ref}_a B \quad r \subseteq a
\] (**) 

However, instead we add a general rule that allows us to denote the reading level of an expression \(E\):

\[
\Sigma; \Gamma \vdash E \Rightarrow_{(r, w)} A \quad r \not\subseteq A\] (35)

where the new judgment \(\vdash A \not\subseteq r\) axiomatizes the idea that values of type \(A\), if they are informative at all, are informative only at level \(r\) or above.\(^2\)

In terms of our new notation, our earlier observations are that \(\vdash 1 \not\subseteq r\) for any \(r\), and \(\vdash \text{ref}_a A \not\subseteq r\) whenever \(r \not\subseteq a\).

3.3 Informative types

We now consider some properties of the new judgment \(\vdash A \not\subseteq a\). Several structural rules for the judgment are immediate. If \(A\) is any type at all, then

\[
\vdash A \not\subseteq \perp
\] (1)

That is, if \(A\) is informative at all, then it’s informative only at \(\perp\) or above. In the interest of brevity, in the sequel we will say “informative only above \(a\)” to mean “informative only at \(a\) and above.”

Also, if \(A\) is informative only above \(a\) and if \(b \subseteq a\), then \(A\) is informative only above \(b\). That is, we may choose to discard some knowledge about when a type is informative

\[
\vdash A \not\subseteq a \quad b \subseteq a
\] (10)

Finally, suppose \(A\) is informative only above \(a\), and \(A\) is informative only above \(b\). Then for any \(r\) if values of type \(A\) are informative to computations that read at \(r\), we know

\(^2\)Informativeness is closely related to protectedness in DCC [1].

We discuss the relationship in Section 6.
that both \( a \subseteq r \) and \( b \subseteq r \). Therefore, for any such \( r \), \( a \sqcup b \subseteq r \). So in fact, \( A \) is informative only above \( a \sqcup b \):

\[
\vdash A \not\in a \quad \vdash A \not\in b \quad \vdash A \not\in a \sqcup b \quad (11)
\]

With the structural rules in place, we may consider each of the types in our language. We should keep in mind, that by adding rules to the judgment \( \vdash A \not\in \perp \) we increase the expressive power of the language by allowing more programs to be well-typed. It is always safe to add more restrictive rules in place of more liberal ones. Below we take the most permissive rules that still maintain non-interference, although it is not clear in all cases that there exist programs which need the added flexibility of certain rules.

A value of type \( \text{bool} \) is informative for any computation at all, since it may be trivially analyzed with a conditional. So aside from the structural axiom \( \vdash A \not\in \perp \), there should be no other rules for \( \text{bool} \). We would give a similar account of other type constructors that may be analyzed by cases. For example sum types \( A + B \) or integers \( \text{int} \).

A value of type \( A \rightarrow B \) is used by applying it to some value and using the result. So \( A \rightarrow B \) is informative exactly when \( B \) is:

\[
\vdash B \not\in a \quad \vdash A \not\in B \not\in a \quad (3)
\]

We have already alluded to one rule for the type \( \text{ref}_a A \):

\[
\vdash \text{ref}_a A \not\in a \quad (5)
\]

which, in combination with the structural rule (10) gives us the upcall rule (***) from Section 3.2.

However there is another rule for refs. Even if a computation can read from a store location of type \( \text{ref}_a A \) (i.e., its operation level is above \( b \)), only if \( A \) is informative at its operation level, can \( \text{ref}_a A \) be informative:

\[
\vdash A \not\in a \quad \vdash \text{ref}_b A \not\in a \quad (6)
\]

Read-only store locations are useful only to computations that may read from them. Consequently, by an argument similar to the one for read-write store cells, we have the two rules:

\[
\vdash \text{ref}_a A \not\in a \quad (7) \quad \vdash \text{ref}_b A \not\in a \quad (8)
\]

For write-only store cells \( \text{ref}_{w_a} A \), we have to consider aliasing. One way that a computation may learn whether two store locations are aliases is by writing a known value to one of them, and then reading out the value from the other. Because of subtyping, if a lower-security computation has a store location \( \ell \) of type \( \text{ref}_a A \), a value of type \( \text{ref}_{w_a} A \) may be informative if the computation can read from (the seemingly unrelated) \( \ell \). As a result, we have the following rule:

\[
\vdash \text{ref}_{w_a} A \not\in a \quad (9)
\]

It is instructive to consider in detail the problem with write-only store locations \( \text{ref}_{w_a} A \). Suppose that instead of the rule (9), we had the following rule:

\[
\vdash \text{ref}_{w_a} A \not\in b \quad (\text{incorrect})
\]

That is, the same as the rule for unit: a value of type \( \text{ref}_{w_a} A \) is only informative above some security level \( b \), for any \( b \), i.e. not informative.

The following computation shows that with the incorrect rule, it is possible to leak high security information (whether the value of secret, a T-security bool, is true) to a low security computation:

\[
\begin{align*}
\text{let } x &= \text{ref}_{\perp} \text{(false : bool)} \text{ in } \\
\text{let } y &= \text{ref}_{\perp} \text{(false : bool)} \text{ in } \\
\text{let } z &= (\text{let } q = \text{!secret} \text{ in } \\
& \text{[if } q \text{ then } x \text{ else } y \text{]} \text{ in } \\
\text{let } w &= \text{!x in } \\
\text{[w]}
\end{align*}
\]

The program lets \( z \) alias either \( x \) or \( y \) depending on the value of secret. The computation whose value is assigned to \( z \) may be subsumed to type \( \text{ref}_{\perp} \text{bool} \), and by the incorrect rule, \( \vdash \text{ref}_{\perp} \text{bool} \not\in T \), so the operation level of that computation can be dropped to \( (\perp, T) \) (and subsumed to \( (\perp, \perp) \)). Then by writing a known value to \( z \), we can observe in another alias of the same location is sufficient to learn about secret. We can give the entire computation the operation level \( (\perp, T) \) while it demonstrably returns the high-security value.

Finally, consider the type \( \text{O}[(r, w)] \ A \). A value of this type is informative both to computations that may read at least security level \( w \) (that is, the level the suspended expression writes to), and to computations for which the type \( A \) is informative:

\[
\vdash A \not\in a \quad \vdash \text{O}[(r, w)] \ A \not\in w \sqcap a \quad (4)
\]

Having added all of these rules to the language, one may wonder whether the language is still secure. Because of the upcall rule (35), we can no longer argue that a computation with operation level \( (r, w) \) does not read values at a security level above \( r \) (it may now do so, provided the reads are not informative). So the simple informal argument at the end of Section 2 no longer applies. We take up the question of non-interference in the next section.

4 Non-interference

Informally, non-interference says that computations that have a low read level do not depend on values in high security store locations. As in similar arguments [19, 18], “low” means below some fixed security level \( \zeta \), and “high” means not below \( \zeta \).

Operationally, the low security sub-computations of a program should behave identically irrespective of the values in the high security store locations. On the other hand, it is alright for high security sub-computations to behave differently depending on values in high security store locations. However once a high security sub-computation completes, the low security behavior should again be identical modulo the parts of the computation state that are “out of view” of the low security part of the program.

Formally, we define an equivalence property of computation states such that two states are equivalent whenever they agree on the “in view” parts of the computation state. Then, in the style of a confluence proof, we show that this equivalence property is preserved under evaluation.
4.1 Equivalence property

We axiomatize the desired property as a collection of equivalence judgments (on states, stores, terms and expressions) that are summarized in Figure 4.

Stores and States  Certainly values in high security store locations are out of view. Less obviously, some values in the low security locations are out of view as well: if a low security store location appears only out of view, its value is also out of view. We parametrize the store equivalence judgment by a set \( U \) of in view store locations. Two (well-typed) stores are equivalent only if their in view values are equivalent:

\[
\vdash H_1 : \Sigma_1 \quad \Sigma_1 \mid U = \Sigma_2 \mid U \\
\Sigma_1; \Sigma_2; \vdash H_1(\ell) \approx_\zeta H_2(\ell) : \Sigma_1(\ell) \\
\vdash (H_1 : \Sigma_1) \approx_{\| U} (H_2 : \Sigma_2) \quad \text{for } \ell \in U
\]  

(58)

Where the notation \( R \mid X \) means \( \{(x,y) \in R \mid x \in X\} \).

For a pair of computation states, only low security locations that are common to both computations are in view. Since allocation does leak information, it is possible for two programs to allocate different low security locations while executing high security sub-computations. However such locations are out of view for the low security sub-computation.

Pairs of computation states are equivalent if their stores are equivalent on the in-view locations, and if they have equivalent expressions:

\[
\vdash \ (H_1 : \Sigma_1) \approx_{\| \text{dom}(H_1) \cap \text{dom}(H_2) \cap |(\zeta)|} (H_2 : \Sigma_2) \\
\Sigma_1; \Sigma_2; \vdash E_1 \approx E_2 \div_\alpha A \\
\vdash (H_1, \Sigma_1, E_1) \approx_\zeta (H_2, \Sigma_2, E_2) \div_\alpha A
\]  

(59)

Where \( |(\zeta)| = \{ \ell \mid \text{Level}(\ell) \subseteq \zeta \} \) is the set of all low security locations.

Terms and Expressions  High security sub-computations of a program may return different values to the low security sub-computations. However, by the upcall rule, the type of those values must be informative only at high security.

Values of a type that is informative only at high security are out of view. As a result, any two values of such a type are equivalent since two such values vacuously agree on their in view parts:

\[
\Sigma_1; \Gamma \vdash V_1 : A \\
\Sigma_2; \Gamma \vdash V_2 : A \\
\vdash A \not\approx a \quad a \not\in \zeta \\
\Sigma_1; \Sigma_2; \Gamma \vdash V_1 \approx_\zeta V_2 : A
\]  

(39)

The remaining rules for term and expression equivalence are congruence rules that merely require corresponding subterms or sub-expressions to be equivalent. They are listed in Appendix C.

4.2 Non-interference theorem

The main result necessary to establish non-interference is the so-called “Hexagon Lemma” (Figure 5): given two equivalent computations states that each take a step, we show that in zero or more steps we can reach two computation states that are again equivalent.

<table>
<thead>
<tr>
<th>Judgment</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Sigma_1; \Sigma_2; \Gamma \vdash M_1 \approx_{\zeta} M_2 : A )</td>
<td>Term Equivalence</td>
</tr>
<tr>
<td>( \Sigma_1; \Sigma_2; \Gamma \vdash E_1 \approx_{\zeta} E_2 \div_\alpha A )</td>
<td>Expression Equivalence</td>
</tr>
<tr>
<td>( \vdash (H_1 : \Sigma_1) \approx_{| U} (H_2 : \Sigma_2) )</td>
<td>Store Equivalence</td>
</tr>
<tr>
<td>( \vdash S_1 \approx_\zeta S_2 \div_\alpha A )</td>
<td>State Equivalence</td>
</tr>
</tbody>
</table>

Figure 4: Equivalence judgments

As previously noted, while a program is executing a high security sub-computation, it may behave differently based on the contents of the high-security store. However, as the following preliminary lemma shows, during any such high security steps, the in view parts of the stores remain equivalent.

**Lemma 4.1 (High Security Step (HSS)).** Given two states \((H_1, \Sigma_1, E_1)\) and \((H_2, \Sigma_2, E_2)\) such that

- \( \vdash (H_1 : \Sigma_1) \approx_{\| U} (H_2 : \Sigma_2) \) where \( U = \text{dom}(\Sigma_1) \cap \text{dom}(\Sigma_2) \cap |(\zeta)| \), and
- for \( i = 1, 2, \) there exist \( C_i \) and \( o_i = (r_i, w_i) \) such that \( \Sigma_i ; \vdash E_i \div_\alpha C_i \) and \( w_i \not\subseteq \zeta \).

If \( (H_1, \Sigma_1, E_1) \rightarrow^* (H'_1, \Sigma'_1, E'_1) \) and \( (H_2, \Sigma_2, E_2) \rightarrow^* (H'_2, \Sigma'_2, E'_2) \) then

\( \vdash (H'_1 : \Sigma'_1) \approx_{\| U'} (H'_2 : \Sigma'_2) \) where \( U' = \text{dom}(\Sigma_1) \cap \text{dom}(\Sigma_2) \cap |(\zeta)| \)

The proof of this lemma appears in the companion technical report [3].

With the preliminaries out of the way, we go on to the core of the proof.

**Lemma 4.2 (Hexagon Lemma).** For all \( \zeta, o = (r, w) \) with \( r \subseteq \zeta \), and if

- \( \vdash S_1 \approx_\zeta S_2 \div_\alpha C \)
- \( S_1 \rightarrow S'_1, S_2 \rightarrow S'_2 \)
- \( S'_1 \downarrow, S'_2 \downarrow \)

then there exist \( S''_1, S''_2 \) such that

- \( S'_1 \rightarrow^* S''_1, S'_2 \rightarrow^* S''_2 \)
- \( \vdash S''_1 \approx_\zeta S''_2 \div_\alpha C \)
Proof. By Inversion on \( \vdash S_1 \approx \llbracket S_2 \rrbracket C \), we get that each computation state \( S_i \) is a triple \( (H_i, \Sigma_i, E_i) \), and that the two stores agree on their common set of labels:

\[
\vdash (H_1 : \Sigma_1) \approx_U^\llbracket (H_2 : \Sigma_2) \rrbracket
\]

where \( U = \text{dom}(\Sigma_1) \cap \text{dom}(\Sigma_2) \cap \|()\| \) and the two expressions are equivalent

\[
\Sigma_1; \Sigma_2; \vdash E_1 \approx \llbracket E_2 \rrbracket \circ \alpha C
\]

We prove the theorem by induction on latter derivation.

We give the most interesting case below. The remaining cases are proved in the companion technical report.

- Case

\[
\frac{\Sigma_1; \Sigma_2; \Gamma \vdash E_1 \approx \llbracket E_2 \rrbracket \circ (r', w) C \vdash C \circ r'}{\Sigma_1; \Sigma_2; \Gamma \vdash E_1 \approx \llbracket E_2 \rrbracket \circ (\llbracket r, w \rrbracket) C}
\] (51)

This is the upcall case. We have two expressions that may read as high as \( r' \), but whatever value they compute is ultimately not informative to any other computation unless it is also above \( r' \).

If \( r' \subseteq \llbracket \xi \rrbracket \), we can invoke the induction hypothesis to get two equivalent computation states with operation level \( (r', w) \), and then use the upcall rule to construct the desired derivation (with operation level \( (r, w) \)).

On the other hand, if \( r' \not\subseteq \llbracket \xi \rrbracket \), then since \( r' \subseteq w \), it follows that \( w \not\subseteq \llbracket \xi \rrbracket \) and so, by the High Security Step Lemma, running both of the computation states to completion produces equivalent stores. Since we also know that their return values are out of view, we can show that the resulting terminal states are equivalent.

\[
\Box
\]

Finally we are ready to prove non-interference. Starting with some initial store \( H \) (well-typed with store type \( \Sigma \)) and an expression to execute \( E \) with a free variable \( x \), if we plug in different values \( V_1, V_2 \) for \( x \), then provided that the in-view parts of \( V_1, V_2 \) are equivalent, we expect that if the resulting programs \( (H, \Sigma, E[V_1/x]) \), \( (H, \Sigma, E[V_2/x]) \) run to termination, the resulting terminal states will be equivalent on their in view parts.

5 The SLam Calculus

Our account differs substantially from prior secure programming languages where each value has a security level. In such languages, terms are classified by security types: pairs of an ordinary type and a security level. The type system ensures that each term is assigned a security level at least as high as the security level of the terms contributing to it. In our account only the store provides security. A natural question is whether we sacrifice expressive power in comparison to value-oriented secure languages.

We will show that our language is at least as expressive by exhibiting an embedding of the SLam calculus [4] into our language. The embedding is not only type correct, but also preserves the security properties of SLam. We assume some familiarity with SLam.

Our presentation of the SLam calculus is based on the purely-functional call-by-value variant presented in Abadi, et al. [1]. The types of SLam include unit, booleans and functions from security annotated values to security annotated results. The syntax of expressions is summarized in Figure 6. In addition to variables, SLam has a security-annotated unit value, and security-annotated boolean values, and functions. Since functions are themselves values, abstractions are annotated not only with the security type of the argument, but also with a security level for the function itself. One additional operation \( \text{protect}, e \) is used to increase the security level of the value of \( e \).

The complete operational semantics and typing rules of SLam are presented in Appendix D.

\[
t \in \text{types} \quad ::= \quad 1 \mid \text{bool} \mid s_1 \rightarrow s_2
\]

\[
s \in \text{security types} \quad ::= \quad (t, a)
\]

\[
\oplus \in \text{arithmetic ops} \quad ::= \quad + \mid \cdots
\]

\[
e \in \text{expressions} \quad ::= \quad x \mid s_a \mid \text{true}_a \mid \text{false}_a
\]

\[
| \quad \text{if } c_1 \text{ then } e_1 \text{ else } c_2
\]

\[
| \quad (\lambda x : s.e)_a
\]

\[
| \quad \text{protect}, e
\]

Figure 6: SLam Calculus Syntax

The idea of the embedding is to translate expressions \( e \) of SLam to expressions \( E \) (not terms) of our language in such a manner that the operation level of \( E \) is \( (\bot, \top) \). That is, the translated expression does not read (informative) values above \( \bot \) nor does it write below \( \top \).

In order to get the desired effect, we store the result of \( e \) in a ref cell. The ref has the effect of sealing the return value of \( e \) at its corresponding security level: values stored in a ref cell are informative only at the security level of the ref. So for example if \( e \) had security type \( \text{bool}, \top \), the result of the translation would be a newly allocated store location of type \( \text{ref} \circ \text{bool} \).

Finally, since a \( \text{ref}_a A \) is informative only above \( a \), we may demote the read level of \( E \) to \( \bot \). And since SLam is purely functional, there are no writes in the translation, so the write level is \( \top \).

Suppose \( x \) is a SLam variable with security type \( \text{bool}, \top \). The SLam expression

\[
\text{if } x \text{ then false}_\top \text{ else true}_\top
\]
negates \( x \). In the translation, \( x \) stands for a \( \top \)-level store location containing that contains a boolean: \( \text{refr}_\top \, \text{bool} \). The expression if \( x \) then false else true is translated into the expression:

\[
\begin{align*}
\text{let } y &= !x \text{ in} \\
\text{let val } y' &= (\text{if } y \text{ then val refr } (\text{false} : \text{bool}) \\
&\quad \text{else val refr } (\text{true} : \text{bool})) \text{ in} \\
[y']
\end{align*}
\]

First \( y \) gets the value stored in \( x \). That sub-expression has operation level \( (\top, \top) \), there so must the other sub-expressions for the translation to be well-typed. Based on the value of \( y \), \( y' \) gets the result of one of the two allocation expressions, and the value of \( y' \) is the result of the whole expression. The entire expression has operation level \( \langle \bot, \bot \rangle \) and return type \( \text{refr}_\top \, \text{bool} \). Since that type is informative only at \( \top \), the operation level may be lowered to \( \langle \bot, \bot \rangle \).

### The Encoding

We define the function \( \tau \) to formalize our embedding of SLam security types:

\[
\begin{align*}
\tau(t, a) &= \text{refr}_\alpha \tau \\
\tau(\top) &= 1 \\
\tau(\text{bool}) &= \text{bool} \\
\tau(s_1 @ s_2) &= \tau \rightarrow \bigcirc(\bot, \bot)^{s_2}
\end{align*}
\]

Since SLam security types are translated to store types in our language, the translation of SLam functions must be able to read from the store. So the translation of the SLam function type has a monadic codomain.

The encoding for SLam expressions is given by a judgment \( \Gamma \vdash e : s \Rightarrow E \). We assume that the metavariable \( y \) stands for variables in our calculus that are not used in SLam expressions. We use \( \text{run} \, M \) as syntactic sugar for let val \( y = M \) in \( [y] \)

\[
\begin{align*}
\Gamma \vdash e : s \Rightarrow E \\
\Gamma \vdash x : \Gamma(x) \Rightarrow [x] \\
\Gamma \vdash s_a : (1, a) \Rightarrow \text{refr}_\alpha (s : 1) \\
\Gamma \vdash \text{true}_a : (\text{bool}, a) \Rightarrow \text{refr}_\alpha (\text{true} : \text{bool}) \\
\Gamma \vdash \text{false}_a : (\text{bool}, a) \Rightarrow \text{refr}_\alpha (\text{false} : \text{bool}) \\
\Gamma \vdash e_1 : (\text{bool}, a) \Rightarrow E_1 \\
\Gamma \vdash e_2 : s \Rightarrow E_2 \\
\Gamma \vdash e_3 : s \Rightarrow E_3 \\
\Gamma \vdash \text{let } y_1 = E_1 \text{ in} \\
\text{let } y_2 = y_1 \text{ in} \\
\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : s \cdot a \Rightarrow \\
\text{run } (\text{if } y_2 \text{ then val } E_2 \text{ else val } E_3) \\
\Gamma, x : s \Rightarrow e : s' \Rightarrow E \\
\Gamma \vdash (\lambda x : s. e)_a : (s \rightarrow s', a) \Rightarrow \text{refr}_\alpha (\lambda x : \pi. \text{val } E : \pi \rightarrow \bigcirc(\bot, \bot)^{s'})
\end{align*}
\]

The proof is given in the companion technical report.

### Security

Of course a type correct (but insecure) embedding could be constructed by ignoring the security levels of the source and placing everything at level \( \bot \). We wish to show that the embedding is actually secure. To do so, we show that the following property, an instance of SLam’s non-interference theorem, is preserved by the embedding:

\[
\text{if } a \not\subseteq b, \text{ and if } \Gamma \vdash f : ((t, a) \rightarrow (\text{bool}, b), b) \text{ and}
\Gamma \vdash e_1, e_2 : (t, a) \text{ then } fe_1 \equiv fe_2
\]

Corresponding to the SLam expression \( f \) is its translation \( F \) in our language, which has type:

\[
\text{refr}_b (\text{refr}_a \tau \rightarrow \bigcirc(\bot, \bot), \text{refr}_b \text{bool})
\]

That is, \( F \) evaluates to a store location that contains a function from store locations (of appropriate type) to computations that return a store location containing a boolean. We would like to show that given different initial locations, the result store locations contain the same boolean value, since the security level \( a \) of the inputs is high in relation to the security level \( b \) of the results.

### Theorem 5.2 (Adequacy of translation)

Suppose \( \vdash f : ((t, a) \rightarrow (\text{bool}, b), b) \Rightarrow F \) and \( a \not\subseteq b \). Let \( H, \Sigma \) be arbitrary such that \( \vdash H : \Sigma \), and let \( \Sigma_i : \vdash e_1, e_2 : \text{refr}_a \tau \). Let

\[
E_i = \begin{cases} 
\text{let } y = F \text{ in} \\
\text{let } z = !y \text{ in} \\
\text{run } (z \ell_i)
\end{cases}
\]

for \( i = 1, 2 \), and suppose

\[
(H, \Sigma, E_1) \Rightarrow^* (H_1, \Sigma_1, [V_1])
\]

and

\[
(H, \Sigma, E_2) \Rightarrow^* (H_2, \Sigma_2, [V_2])
\]

then \( V_1 = \ell_1 \) and \( V_2 = \ell_2 \) and \( H_1(\ell_1) = H_2(\ell_2) \)

Proof. By the type-correctness of the translation,

\[
\{ \} ; \vdash F \Rightarrow^* \text{refr}_b (\text{refr}_a \tau \rightarrow \bigcirc(\bot, \bot), \text{refr}_b \text{bool})
\]

10
By store extension and typing rules,
\[
\Sigma_i; \vdash E_i \sim_{(\bot, \top)} \text{refr}_b \text{bool}
\]
for \(i = 1, 2\) Note that since \(a \not\approx b\),
\[
\Sigma; \Sigma_i; \vdash \ell_1 \approx_b \ell_2 : \text{refr}_a \top
\]
All the other subterms of \(E_1\) and \(E_2\) (for example, \(F. !y\), etc) are the same, so by reflexivity, and various equivalence rules, it’s easy to establish a derivation for
\[
\vdash (H, \Sigma, E_1) \approx_b (H, \Sigma, E_2) \sim_{(\bot, \top)} \text{refr}_b \text{bool}
\]
By non-interference,
\[
\vdash (H_1, \Sigma_1, [V_1]) \approx_b (H_2, \Sigma_2, [V_2]) \sim_{(\bot, \top)} \text{refr}_b \text{bool}
\]
From which it follows for \(i = 1, 2\) that \(V_i = \ell_i\) by a Canonical Forms lemma, and \(\text{Level}(\ell_i) \subseteq b\) by inversion. By inversion on the equivalence derivations, it follows that \(\ell_1 = \ell_2\) and by inversion on the store equivalence, it follows that
\[
\Sigma_1; \Sigma_2; \vdash H_1(\ell_1) \approx H_2(\ell_2) : \text{bool}
\]
And since \(\text{bool}\) is informative only above \(\bot\), by inversion on the equivalence, \(H_1(\ell_1) = H_2(\ell_2)\) \(\square\).

The embedding above deals with functional SLam for simplicity, but a similar embedding can also be contrived for effectful languages. In the companion technical report [3] we give a similar embedding for an imperative version of SLam [15].

6 Related Work

There is a large body of existing work on type systems for secure information flow. Volpano, Smith and Irvine [14] first showed how to formulate an information flow analysis as a type system. An excellent survey by Sabelfeld and Myers [13] outlines the key ideas in the design of secure programming languages.

Prior work on secure languages with imperative features, such as Pottier and Simonet’s work on core-ML [11, 12] take the side-effect view of computations: a term of any type \(A\) may have a side-effect. In contrast, we have taken a monadic view of computation: only expressions may have an effect.

Our account is most related to the Dependency Core Calculus of Abadi, Banerjee, et al. [1]. Like our language, DCC uses a family of monads to reason about information flow. However in DCC, terms of monadic type are used to seal up values at a security level. In our account, monads are used in a more traditional role as a means of threading state through a program.

Central to DCC is the notion of \textit{protectedness} of a type at a security level. If \(T\) is protected at \(a\) then \(T\) is at least as secure as \(a\). This is closely related to our notion of informativeness. (Also, a similar notion called “tampering levels” appears in Honda and Yoshida [5].)

When viewed through the lens of the encoding of SLam, the two relations serve the same purpose, ensuring that a computation’s output is at least as secure as its inputs. In DCC, this is done directly. In our account, this occurs indirectly: to access a value carrying information only at a particular level, a computation must adopt a read level at least as high. (However, our account also offers the facility — not employed in the SLam embedding — not to seal all computations’ return values in order to obtain a \(\bot\) effective read level).

The definitions of protectedness and informativeness are the same on the standard type operators, but do not include the idiosyncratic cases: our language has no analog of DCC’s monad, nor does DCC contain references or a traditional \((\text{i.e., effects-oriented})\) monad. Moreover, if it did, we conjecture that DCC’s definition for these would be somewhat different from ours.

Nevertheless, the similarity between the two suggests that our account might be profitably combined with DCC to produce a language capable of expressing security in both value-oriented and store-oriented fashions.

7 Conclusion

We give an account of secure information flow in the context of a higher-order language with mutable state. Moreover, motivated by the logical underpinnings of computation, we arrive at an store-oriented approach to security. Rather than sealing values at a security level, we instead associate security with the store. A family of monadic types is used to keep track of the effects of computations. To account for upcalls, we classify the informativeness of types at particular security levels. Since we treat terms apart from the effectful expressions, our approach can straightforwardly encompass additional type constructors.

Our formulation of the monadic language is in the style of Pfenning and Davies [9]. One avenue of future work is to study whether there is a formulation of information flow in a modal logic that decomposed our monad into the possibility and necessity modalities.

Several questions remain about our language. In the case that the lattice of security levels \(\mathcal{L}\) is finite, it is obvious that the informativeness judgment \(\vdash A \sim a\) is decidable. Although it is reasonable to assume that in a practical secure programming system, there would only be a finite number of security levels, it is nonetheless interesting to know whether \(\vdash A \sim a\) is decidable even in the general case where \(\mathcal{L}\) may be infinite.

A general open problem in the area of secure programming languages is how to devise a type system for a language with \textit{declassification} operations. Declassification occurs when a low-security computation makes use of a high-security value, but in a way such that the information gained from the high-security value is deemed an acceptable leak. Recently, Zdancewic and Myers [17] show how to characterize so-called \textit{robust declassification} in programs such that an attacker may observe the declassified values, but may not exploit them to gain additional high-security information. Zdancewic [16] then gives a type system for robust declassification. Since declassification is fundamentally an operation, we conjecture that our store-oriented viewpoint could be meshed with Zdancewic and Myers to provide a logic of declassification.

Acknowledgments Thanks to Matthew Harren and Steve Zdancewic for their comments and suggestions.
References


A Informativeness judgment

\[ \vdash A \not\to \perp \]  
(1)

\[ \vdash A \not\to a \]  
(2)

\[ \vdash B \not\to a \]  
(3)

\[ \vdash A \not\to a \]  
(4)

\[ \vdash \text{ref}_b A \not\to b \]  
(5)

\[ \vdash A \not\to a \]  
(6)

\[ \vdash \text{ref}_b A \not\to a \]  
(7)

\[ \vdash A \not\to a \]  
(8)

\[ \vdash \text{ref}_w A \not\to a \]  
(9)

\[ \vdash A \not\to a \]  
(10)

\[ \vdash A \not\to a \]  
(11)
B Typing Judgment rules

\[ \vdash A \leq B \]

\[ \vdash A \leq A \quad (12) \]

\[ \vdash A \leq A' \vdash B' \leq B \]

\[ \vdash A' \rightarrow B' \leq A \rightarrow B \quad (13) \]

\[ \vdash A \leq B \quad o \leq o' \]

\[ \vdash \bigcirc_o A \leq \bigcirc_o B \quad (14) \]

\[ \vdash A \leq B \quad a \subseteq b \]

\[ \vdash \text{ref}_A A \leq \text{ref}_A B \quad (15) \]

\[ \vdash B \leq A \quad b \subseteq a \]

\[ \vdash \text{ref}_A A \leq \text{ref}_A B \quad (16) \]

\[ \vdash A \leq B \quad a \subseteq b \]

\[ \vdash \text{ref}_A A \leq \text{ref}_A B \quad (17) \]

\[ \vdash B \leq A \quad b \subseteq a \]

\[ \vdash \text{ref}_A A \leq \text{ref}_A B \quad (18) \]

\[ \Sigma; \Gamma \vdash M : A \]

\[ \Sigma; \Gamma \vdash \cdot : 1 \quad (19) \]

\[ \Sigma; \Gamma \vdash x : \Gamma(x) \quad (20) \]

\[ \Sigma; \Gamma \vdash \text{true} : \text{bool} \quad (21) \]

\[ \Sigma; \Gamma \vdash \text{false} : \text{bool} \quad (22) \]

\[ \Sigma; \Gamma \vdash M : \text{bool} \quad \Sigma; \Gamma \vdash N_1 : A \quad \Sigma; \Gamma \vdash N_2 : A \quad (23) \]

\[ \Sigma; \Gamma \vdash \text{if } M \text{ then } N_1 \text{ else } N_2 : A \]

\[ \Sigma; \Gamma \vdash \ell : \text{ref}_{\text{Level}(\ell)} \Sigma(\ell) \quad (24) \]

\[ \Sigma; \Gamma, x : A \vdash M : B \quad (25) \]

\[ \Sigma; \Gamma \vdash \lambda x : A.M : A \rightarrow B \]

\[ \Sigma; \Gamma \vdash M : A \rightarrow B \quad \Sigma; \Gamma \vdash N : A \quad (26) \]

\[ \Sigma; \Gamma \vdash M N : B \]

\[ \Sigma; \Gamma \vdash E : \bigcirc_o A \quad (27) \]

\[ \Sigma; \Gamma \vdash \text{val } E : \bigcirc_o A \]

\[ \Sigma; \Gamma \vdash M : A \quad A \leq B \]

\[ \Sigma; \Gamma \vdash M : B \quad (28) \]

\[ \Sigma; \Gamma \vdash E : \bigcirc_o A \]

\[ \Sigma; \Gamma \vdash M : A \quad (29) \]

\[ \Sigma; \Gamma \vdash [M] : \bigcirc_{(\ell, T)} A \]

\[ \Sigma; \Gamma \vdash x : A \vdash E : \bigcirc_o B \]

\[ \Sigma; \Gamma \vdash \text{let val } x = M \text{ in } E : \bigcirc_o B \quad (30) \]

\[ \Sigma; \Gamma \vdash M : A \]

\[ \Sigma; \Gamma \vdash \text{ref}_A (M : A) : A \quad \text{ref}_A A \quad (31) \]

\[ \Sigma; \Gamma \vdash M : \text{ref}_A A \quad \Sigma; \Gamma \vdash N : A \]

\[ \Sigma; \Gamma \vdash M := N : A \quad (32) \]

\[ \Sigma; \Gamma \vdash E : \bigcirc_o A \quad o' \leq o \quad (33) \]

\[ \Sigma; \Gamma \vdash E : \bigcirc_o A \quad \Sigma; \Gamma \vdash E : \bigcirc_o A \]

\[ \Sigma; \Gamma \vdash E : \bigcirc_o A \quad B \leq C \quad (34) \]

\[ \Sigma; \Gamma \vdash H : \Sigma \]

\[ \text{dom}(\Sigma) = \{\ell_1, \ldots, \ell_n\} \quad \Sigma; \vdash V_1 : \Sigma(\ell_1) \text{ for } 1 \leq i \leq n \quad (37) \]

\[ \Sigma; \Gamma \vdash \text{true} : \text{bool} \]

\[ \Sigma; \Gamma \vdash \text{false} : \text{bool} \]

\[ \Sigma; \Gamma \vdash E : \bigcirc_o A \]

\[ \Sigma; \Gamma \vdash H : \Sigma \quad \Sigma; \vdash E : \bigcirc_o A \quad (38) \]

C Equivalent View Judgments

\[ \Sigma_1; \Sigma_2; \Gamma \vdash M_1 \simeq_\zeta M_2 : A \]

\[ \vdash A \not\sim a \quad a \not\in \zeta \quad \Sigma_1; \Gamma \vdash V_1 : A \quad \Sigma_2; \Gamma \vdash V_2 : A \quad (39) \]

\[ \Sigma_1; \Sigma_2; \Gamma \vdash V_1 \simeq_\zeta V_2 : A \]

\[ \Sigma_1; \Sigma_2; \Gamma \vdash \ast \simeq_\zeta \ast : 1 \quad (40) \]

\[ \Sigma_1; \Sigma_2; \Gamma \vdash x \simeq_\zeta x : \Gamma(x) \quad (41) \]

\[ \Sigma_1; \Sigma_2; \Gamma \vdash \text{true} \simeq_\zeta \text{true} : \text{bool} \quad (42) \]

\[ \Sigma_1; \Sigma_2; \Gamma \vdash \text{false} \simeq_\zeta \text{false} : \text{bool} \quad (43) \]

\[ \Sigma_1; \Sigma_2; \Gamma \vdash M_1 \simeq_\zeta M_2 : \text{bool} \quad \Sigma_1; \Sigma_2; \Gamma \vdash N_1 \simeq_\zeta N_2 : A \quad \Sigma_1; \Sigma_2; \Gamma \vdash P_1 \simeq_\zeta P_2 : A \quad (44) \]

\[ \Sigma_1; \Sigma_2; \Gamma \vdash \text{if } M_1 \text{ then } N_1 \text{ else } P_1 \simeq_\zeta \text{if } M_2 \text{ then } N_2 \text{ else } P_2 : A \]
\[ \Sigma_1; \Sigma_2; \Gamma, x : A \vdash M_1 \approx_\zeta M_2 : B \]
\[ \vdash (H_1 : \Sigma_1) \approx_\zeta \langle \dom(\Sigma_1) \rangle \cup \langle \dom(\Sigma_2) \rangle \cup \langle \zeta \rangle (H_2 : \Sigma_2) \]

**D** The SLam Calculus

**D.1** Operational semantics

\[ e \rightarrow e' \]

- if \( e_1 \) then \( e_2 \) else \( e_3 \) if \( e' \) then \( e_2 \) else \( e_3 \)
- if \( true_a \) then \( e_2 \) else \( e_2 \) else \( e_3 \)
- if \( e_1 \rightarrow e_1' \)
- if \( e_2 \rightarrow e_2' \)
- \( \lambda x : s.e) \rightarrow v \rightarrow e[v/x] \]

**D.2** Typing rules

\[ s \leq s' \]

\[ s_1 \leq s_2 \quad s_2 \leq s_3 \quad s_1 \leq s_3 \]

\[ a \sqsubseteq a' \quad (1, a) \leq (1, a') \quad (\text{bool}, a) \leq (\text{bool}, a') \]

\[ s'_1 \leq s_1 \quad s_2 \leq s_3 \quad a \sqsubseteq a' \quad (s_1 \rightarrow s_2, a) \leq (s'_1 \rightarrow s'_2, a') \]

\[ \Gamma \vdash e : s \]

\[ \Gamma \vdash x : \Gamma(x) \]

\[ \Gamma \vdash \text{true}_a : (\text{bool}, a) \]

\[ \Gamma \vdash \text{false}_a : (\text{bool}, a) \]

\[ \Gamma \vdash e_1 : (\text{bool}, a) \quad \Gamma \vdash e_2 : s \quad \Gamma \vdash e_3 : s 
\]

\[ \Gamma \vdash \text{if} e_1 \text{ then } e_2 \text{ else } e_3 : s \]

\[ \Gamma, x : s_1 \vdash e : s_2 \]

\[ \Gamma \vdash (\lambda x : s_1.e) a : (s_1 \rightarrow s_2, a) \]

\[ \Gamma \vdash e_1 : (s_1 \rightarrow s, a) \quad \Gamma \vdash e_2 : s_1 \]

\[ \Gamma \vdash e_1, e_2 : s \bullet a \]

\[ \Gamma \vdash \text{protect}_a e : s \bullet a \]

\[ \Gamma \vdash e : s \quad s \leq s' \]

\[ \Gamma \vdash e : s' \]