A Monadic Analysis of Information Flow Security with Mutable State

Karl Crary* Aleksey Kliger Frank Pfenning
{crary,aleksey,fp}@cs.cmu.edu
Carnegie Mellon University

July 13, 2004

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 elements of the mutable store. This view matches closely with the operational semantics of low-level imperative languages where information flow is expressed by operations on the store. 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 type systems 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, 9, 11, 17].

However, when adopting one of these languages to the Typed Assembly Language [8] setting, one faces a tension between the high-level view of information flowing from the values of sub-terms to the result value of a complete term and the assembly-language imperative view of instructions operating on a mutable store. What is needed is a typed calculus in which values have structure (*i.e.*, like in high level languages) but information flows through the store (*i.e.*, like in a low-level language).

In this paper, we explore this *store-oriented* view of information flow: one of the steps towards a TAL with information flow, we look at a language with a clean separation between values and computations. A suitable starting point is Moggi's monadic metalanguage [6, 7] and its corresponding logic (via a Curry-Howard isomorphism).

Our presentation of lax logic is based on that of Pfenning and Davies [10]. 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 our work inherits these properties, they are not particularly important here. However the term/expression distinction also provides a clean separation between the pure and effectful parts of our analysis, which greatly simplifies our system.

*This material is based on work supported in part by NSF grants CCR-9984812 and CCR-0121633. Any opinions, findings, and conclusions or recommendations in this publication are those of the authors and do not reflect the views of this agency.

Our system bears some resemblance to the work of Abadi *et al.* [1], who also use a monadic structure to reason about information flow. However whereas we use monads in a conventional manner to separate values from computations, they use a monad to endow values with a security level. It is not clear how to adopt their work to a low-level setting where the values and operations ought to correspond to those of a real machine.

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 A is a type, Γ is the usual context and Σ 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 of information flow. The judgment takes the form: $\Sigma; \Gamma \vdash E \div_{(r,w)} A$ indicating that r is an 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 $r \sqsubseteq w$, or else E could manifestly be leaking information.

Our language can be seen as a conservative extension to purely functional languages such as Haskell. Existing terms continue to be type-safe. On the other hand new effectful code that makes use of the security discipline can be cleanly separated.

In lax logic, expressions are internalized as terms using the monadic type $\bigcirc A$. Terms of type $\bigcirc A$ are suspended expressions 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 $\vdash A \nearrow a$ 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 provides at least the expressive power of value-oriented accounts by embedding one value-oriented language 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

$$
\begin{array}{rcll}
A, B, C & \in & \textit{types} & ::= \quad 1 \mid \mathsf{bool} \mid A \to B \mid \mathsf{ref}_a\, A \mid \mathsf{refr}_a\, A \mid \mathsf{refw}_a\, A \mid \bigcirc_o A \\
M, N & \in & \textit{terms} & ::= \quad x \mid * \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ M\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2 \mid \lambda x : A.M \mid MN \mid \ell \mid \mathsf{val}\ E \\
E, F & \in & \textit{expressions} & ::= \quad [M] \mid \mathsf{let\ val}\ x = M\ \mathsf{in}\ E \mid \mathsf{ref}_a\,(M : A) \mid\ !M \mid M := N \\
\Gamma & \in & \textit{contexts} & ::= \quad \cdot \mid \Gamma, x : A \\
\Sigma & \in & \textit{store types} & ::= \quad \{\} \mid \Sigma\{\ell : A\} \\
V & \in & \textit{values} & ::= \quad * \mid \mathsf{true} \mid \mathsf{false} \mid \lambda x : A.M \mid \ell \mid \mathsf{val}\ E \\
H & \in & \textit{stores} & ::= \quad \{\} \mid H\{\ell \mapsto V\} \\
S & \in & \textit{states} & ::= \quad (H, \Sigma, E)
\end{array}
$$

Figure 1: Syntax

be practical. By including some additional rules, we will be able to make it more useful while still retaining the required secrecy property.

As in other work on information flow, we have in mind an arbitrary lattice $(\mathcal{L}, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ of security levels.

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 may obviously be insecure, henceforth we restrict our attention only to operation levels (r, w) with $r \sqsubseteq w$.

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 $r \sqsubseteq r'$. Similarly, if it writes at level at least w, then it writes at level at least w', provided that $w' \sqsubseteq w$. That is, operation levels are covariant in the reads and contravariant in the writes: $(r, w) \preceq (r', w')$ iff $(r \sqsubseteq r'$ and $w' \sqsubseteq w)$

2.1 Syntax

The full syntax of our language is given in Figure 1. The language is split into two syntactic categories: pure terms M that are evaluated to values V and expressions E that are executed for effect as part of compuation states S.

Terms At the term level, we have variables, unit, booleans and conditional terms, function abstractions and applications. For simplicity, we did not include a mechanism for defining recursive terms, although the inclusion of such a facility would not pose a problem. Store locations are also terms, with each location ℓ having a fixed security level $\mathsf{Level}(\ell)$. The store associates locations with the values they contain. A subtyping relation, allows us to treat store cells as either read-write, read-only, or write-only. The term $\mathsf{val}\ E$ allows expressions to be included at the term level as an element of the monadic type $\bigcirc_o A$. Since terms are pure, a $\mathsf{val}\ E$ does not execute the expression E, but rather represents a suspended computation.

Expressions The expressions include a trivial return expression $[M]$. 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 $[M]$ is $(\bot, \top)$. Note that $(\bot, \top)$ is the least element in the $\preceq$ ordering, so our subsumption principle will let us weaken the operation level of $[M]$ to any operation level.

The sequencing expression $\mathsf{let\ val}\ x = M\ \mathsf{in}\ F$ evaluates M down to some $\mathsf{val}\ E$, 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 $\mathsf{let}\ x = E\ \mathsf{in}\ F$ as syntactic sugar for $\mathsf{let\ val}\ x = \mathsf{val}\ E\ \mathsf{in}\ F$, and $\mathsf{run}\ M$ for $\mathsf{let\ val}\ y = M\ \mathsf{in}\ [y]$.

$\boxed{\Sigma; \Gamma \vdash M : A}$

$$\frac{}{\Sigma; \Gamma \vdash x : \Gamma(x)}\ (1) \quad \frac{}{\Sigma; \Gamma \vdash \ell : \mathsf{ref}_{\mathsf{Level}(\ell)}\, \Sigma(\ell)}\ (2) \quad \frac{\Sigma; \Gamma \vdash M : \mathsf{bool} \quad \Sigma; \Gamma \vdash N_1 : A \quad \Sigma; \Gamma \vdash N_2 : A}{\Sigma; \Gamma \vdash \mathsf{if}\ M\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2 : A}\ (3)$$

$$\frac{\Sigma; \Gamma, x : A \vdash M : B}{\Sigma; \Gamma \vdash \lambda x : A.M : A \to B}\ (4) \quad \frac{\Sigma; \Gamma \vdash M : A \to B \quad \Sigma; \Gamma \vdash N : A}{\Sigma; \Gamma \vdash M\ N : B}\ (5) \quad \frac{\Sigma; \Gamma \vdash E \div_o A}{\Sigma; \Gamma \vdash \mathsf{val}\ E : \bigcirc_o A}\ (6) \quad \frac{\Sigma; \Gamma \vdash M : A \quad \vdash A \leq B}{\Sigma; \Gamma \vdash M : B}\ (7)$$

Figure 2: Typing rules (terms).

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

Allocation cannot leak information in our setting. 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 not aliased. Thus, there can be no implicit flow of information via an allocation expression. As a result, allocation has operation level $(\bot, \top)$. Of course if there were a primitive mechanism in place to distinguish locations (for example by comparing locations for equality), allocation would once again be observable.

Although there is not a primitive mechanism for recursion at the level of expressions, recursion can be encoded at the level of expressions using back-patching, see an example in Section 3.2.

States A computation state is a partially executed program, and consists of a triple (H, Σ, E) of a store H, a store type Σ 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, Σ, E), the store binds occurrences of store locations ℓ in H and E, and we identify computation states up to level-preserving renaming of store locations. In addition, as usual, we identify all constructs up to renaming of bound variables.

2.2 Static Semantics

The type system of our language consists of two main mutually recursive judgments for typing terms and expressions, and some judgments for typechecking stores, and computation states. The first judgment $\Sigma; \Gamma \vdash M : A$ says that the term M has type A in the context Γ, where the store has type Σ. The jugment for expressions $\Sigma; \Gamma \vdash E \div_o A$ says that E returns a value of type A and performs only operations within level o.

We assume that contexts Γ 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 ℓ.

Terms The typing rules for terms are unsurprising for a simply-typed lambda calculus with unit, bool and function types. A store location ℓ (provided that it is in $\mathrm{dom}(\Sigma)$) has type $\mathsf{ref}_{\mathsf{Level}(\ell)}\Sigma(\ell)$. A computation term $\mathsf{val}\ E$ has the type $\bigcirc_o A$, provided the expression E has type A and operation level o. The rules are summarized in figure 2.

Expressions The typing rules for expressions (given in Figure 3) follow our informal description. Trivial computations have the type of their return value, and operation level $(\bot, \top)$ (rule 8). By rule (9), the sequencing

$\boxed{\Sigma; \Gamma \vdash E \div_o A}$

$$\frac{\Sigma; \Gamma \vdash M : A}{\Sigma; \Gamma \vdash [M] \div_{(\bot,\top)} A}\ (8) \quad \frac{\Sigma; \Gamma \vdash M : \bigcirc_o A \quad \Sigma; \Gamma, x : A \vdash E \div_o A}{\Sigma; \Gamma \vdash \mathsf{let\ val}\ x = M\ \mathsf{in}\ E \div_o A}\ (9) \quad \frac{\Sigma; \Gamma \vdash M : A}{\Sigma; \Gamma \vdash \mathsf{ref}_a\ (M : A) \div_{(\bot,\top)} \mathsf{ref}_a\ A}\ (10)$$

$$\frac{\Sigma; \Gamma \vdash M : \mathsf{refr}_a\ A}{\Sigma; \Gamma \vdash !M \div_{(a,\top)} A}\ (11) \quad \frac{\Sigma; \Gamma \vdash M : \mathsf{refw}_a\ A \quad \Sigma; \Gamma \vdash N : A}{\Sigma; \Gamma \vdash M := N \div_{(\bot,a)} 1}\ (12) \quad \frac{\Sigma; \Gamma \vdash E \div_o A \quad o \preceq o'}{\Sigma; \Gamma \vdash E \div_{o'} A}\ (13) \quad \frac{\Sigma; \Gamma \vdash E \div_o A \quad \vdash A \leq B}{\Sigma; \Gamma \vdash E \div_o A}\ (14)$$

Figure 3: Typing rules (expressions).

$\boxed{\vdash A \leq B}$

$$\frac{\vdash A \leq B \quad a \sqsubseteq b}{\vdash \mathsf{refr}_a\ A \leq \mathsf{refr}_b\ B}\ (15) \quad \frac{\vdash B \leq A \quad b \sqsubseteq a}{\vdash \mathsf{refw}_a\ A \leq \mathsf{refw}_b\ B}\ (16) \quad \frac{\vdash A \leq B \quad a \sqsubseteq b}{\vdash \mathsf{ref}_a\ A \leq \mathsf{refr}_b\ B}\ (17) \quad \frac{\vdash B \leq A \quad b \sqsubseteq a}{\vdash \mathsf{ref}_a\ A \leq \mathsf{refw}_b\ B}\ (18) \quad \frac{\vdash A \leq B \quad o \preceq o'}{\vdash \bigcirc_o A \leq \bigcirc_{o'} B}\ (19)$$

Figure 4: Selected subtyping rules.

expression $\mathsf{let\ val}\ x = M\ \mathsf{in}\ E$ is well-typed provided both of the sub-computations have the same operation level (which may require using rule (13) to weaken the operation level of the sub-computations). Allocation (rule 10) returns a new read/write store location. For read and write expressions (rules 11 and 12) we only require that the corresponding store location is readable or writable, respectively.

Subtyping Subsumption (rules 7, 14) allows us to weaken the type A of a term M or an expression E, provided A is a subtype of B. Selected subtyping rules are given in Figure 4.

Stores and states A store H is well-typed with store type Σ, provided that each value V_i in the store is well typed under Σ and the empty context, where Σ has the same domain as H. A computation state (H, Σ, E) is well-typed provided that the store and the expression are each well-typed with the same store type:

$$\frac{\mathrm{dom}(\Sigma) = \{\ell_1, \ldots, \ell_n\} \quad \Sigma; \cdot \vdash V_i : \Sigma(\ell_i)\ \text{ for } 1 \leq i \leq n}{\vdash \{\ell_1 \mapsto V_1, \ldots \ell_n \mapsto V_n\} : \Sigma}\ (20) \quad \frac{\vdash H : \Sigma \quad \Sigma; \cdot \vdash E \div_o A}{\vdash (H, \Sigma, E) \div_o A}\ (21)$$

2.3 Operational Semantics and Safety

A computation state is called *terminal* if it is of the form $(H, \Sigma, [V])$. An evaluation relation $S \to S'$ gives the small-step operational semantics for computation states. We write $S \downarrow$ if for some terminal state S', $S \to^* S'$. Since terms are pure, their evaluation rules may be given simply by the relation $M \to M'$ (no store is required). The evaluation rules for terms are entirely standard, and are omitted. The evaluation rules for expressions are given in Figure 5. 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 ℓ.

A computation state S is *stuck* if it is not terminal and there is no S' such that $S \to S'$. In the extended version of the paper [2] we show the expected type safety theorem: whenever S is well-typed, if $S \to^* S'$, then S' is not stuck.

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

$\boxed{S \to S'}$

$$\frac{M \to M'}{(H, \Sigma, \mathsf{let\ val}\ x = M\ \mathsf{in}\ E) \to (H, \Sigma, \mathsf{let\ val}\ x = M'\ \mathsf{in}\ E)}\ \textsc{Letval1} \qquad \frac{(H, \Sigma, E) \to (H', \Sigma', E')}{(H, \Sigma, \mathsf{let\ val}\ x = \mathsf{val}\ E\ \mathsf{in}\ F) \to (H', \Sigma', \mathsf{let\ val}\ x = \mathsf{val}\ E'\ \mathsf{in}\ F)}\ \textsc{Letvalval} \qquad \frac{}{(H, \Sigma, \mathsf{let\ val}\ x = \mathsf{val}\ [V]\ \mathsf{in}\ E) \to (H, \Sigma, E[V/x])}\ \textsc{Letval}$$

$$\frac{\ell \notin \mathrm{dom}(H) \quad \mathsf{Level}(\ell) = a}{(H, \Sigma, \mathsf{ref}_a\,(V : A)) \to (H\{\ell \mapsto V\}, \Sigma\{\ell : A\}, [\ell])}\ \textsc{Ref} \qquad \frac{}{(H, \Sigma, !\ell) \to (H, \Sigma, [H(\ell)])}\ \textsc{Bang} \qquad \frac{\ell \in \mathrm{dom}(H)}{(H, \Sigma, \ell := V) \to (H\{\ell \mapsto V\}, \Sigma, [*])}\ \textsc{Assn} \qquad \frac{M \to M'}{(H, \Sigma, [M]) \to (H, \Sigma, [M'])}\ \textsc{Ret1}$$

Figure 5: Operational Semantics (Expressions, selected rules)

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 $\mathsf{let}\ z = P\ \mathsf{in}\ E$ where $P \div_{(\top,\top)} 1$ and E has operation level $(\bot, \bot)$. P does not leak information because 1 carries no useful information, and P's writes are above E's reading level. Thus we would like to give the entire program the operation level $(\bot, \bot)$. 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 — our notion of *informativeness* — discussed in Section 3.2. We take up the question of non-interference in Section 4.

3.1 A more general example

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

Suppose that $r \sqsubseteq a$. In that case, any computation that may read the $\mathsf{ref}_a\, B$ 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 $r \not\sqsubseteq a$. The particular value of type $\mathsf{ref}_a\, B$ that E returns may carry information from store locations at security level r. For example, E may return one of two such store locations ℓ_1 or ℓ_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 $r \not\sqsubseteq a$, this represents a violation of secure information flow.

So if E returns a $\mathsf{ref}_a\, B$, we can demote its reading level whenever $r \sqsubseteq a$, 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 $\mathsf{ref}_a\, B$ is informative only to computations that may read at least at some security level (namely a) above r.

This observation suggests a new subsumption rule for expressions that alters the operation level:

$$\frac{\Sigma; \Gamma \vdash E \div_{(r,w)} A \quad \vdash A \nearrow r}{\Sigma; \Gamma \vdash E \div_{(\bot,w)} A} \quad (22)$$

where the new *informativeness* judgment $\vdash A \nearrow r$ formalizes the idea that values of type A, if they are informative at all, are informative only at level r or above.[1]

In terms of this new judgment, our earlier observations are that $\vdash 1 \nearrow r$ for any r, and $\vdash \mathsf{ref}_a\, A \nearrow r$ whenever $r \sqsubseteq a$.

[1] Informativeness is closely related to *protectedness* in DCC [1] and to the tampering levels of [5]. We discuss the relationship in Section 6.

$\boxed{\vdash A \nearrow a}$

$$\frac{}{\vdash A \nearrow \bot}\ (23) \quad \frac{\vdash A \nearrow a \quad b \sqsubseteq a}{\vdash A \nearrow b}\ (24) \quad \frac{\vdash A \nearrow a \quad \vdash A \nearrow b}{\vdash A \nearrow a \sqcup b}\ (25) \quad \frac{\vdash B \nearrow a}{\vdash A \to B \nearrow a}\ (26) \quad \frac{}{\vdash \mathsf{ref}_a\, A \nearrow a}\ (27)$$

$$\frac{\vdash A \nearrow a}{\vdash \mathsf{ref}_b\, A \nearrow a}\ (28) \quad \frac{\vdash A \nearrow a}{\vdash \mathsf{refr}_b\, A \nearrow a}\ (29) \quad \frac{}{\vdash \mathsf{refr}_b\, A \nearrow b}\ (30) \quad \frac{}{\vdash \mathsf{refw}_a\, A \nearrow a}\ (31) \quad \frac{\vdash A \nearrow a}{\vdash \bigcirc_{(r,w)} A \nearrow w \sqcap a}\ (32)$$

Figure 6: Informativeness judgment.

$\lambda c : \bigcirc_{(\top,\top)}\mathsf{bool}.$
$\mathsf{val}\ \mathsf{let}\ \textit{wref} = \mathsf{ref}_\top\ (\mathsf{val}\ [*] : \bigcirc_{(\bot,\top)} 1)\ \mathsf{in}$
$\quad \mathsf{let}\ w = [\mathsf{val}\ (\mathsf{let}\ b = \mathsf{run}\ c\ \mathsf{in}\ \mathsf{run}\ (\mathsf{if}\ b\ \mathsf{then}\ \mathsf{val}\ (\mathsf{let}\ w' = !\textit{wref}\ \mathsf{in}\ \mathsf{run}\ w')\ \mathsf{else}\ \mathsf{val}\ [*]))]\ \mathsf{in}$
$\quad \mathsf{let}\ \_ = \textit{wref} := w\ \mathsf{in}$
$\quad \mathsf{run}\ w$

Figure 7: Using rule (22), *untilFalse* has type $\bigcirc_{(\top,\top)}\mathsf{bool} \to \bigcirc_{(\bot,\top)} 1$

3.2 Informativeness

We now consider some properties of the new judgment $\vdash A \nearrow a$ (see Figure 6). Several structural rules (23,24,25) for the judgment are immediate. If A is informative at all, then it's informative only at $\bot$ or above. Also, if A is informative only at or above a and if $b \sqsubseteq a$, then A is informative only at or above b. That is, we may choose to discard some knowledge about when a type is informative. 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 that both $a \sqsubseteq r$ and $b \sqsubseteq r$. Therefore, for any such r, $a \sqcup b \sqsubseteq r$. So, A is informative only above $a \sqcup b$.

A value of type bool is informative for any computation at all, since it may be trivially analyzed with a conditional. So aside from the structural axiom $\vdash A \nearrow \bot$, there should be no other rules for bool. We would give a similar account of other types that may be analyzed by branching (*e.g.*, sum types $A + B$ or integers int).

Since functions are used by application, a value of type $A \to B$ is useful exactly when B is.

We have already alluded to rule (27) for $\mathsf{ref}_a\, A$. There is an additional rule for references. Even if a computation can read from a store location of type $\mathsf{ref}_b\, A$ (*i.e.*, its read level is above b), only if A is informative at its operation level, can $\mathsf{ref}_b\, A$ be informative.

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, they have analogous rules.

For write-only store cells $\mathsf{refw}_a\, A$, 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 ℓ of type $\mathsf{refr}_a\, A$, a value of type $\mathsf{refw}_a\, A$ may be informative if the computation can read from (the seemingly unrelated) ℓ.

Finally, consider the type $\bigcirc_{(r,w)} A$. 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.

With informativeness in hand, many more useful terms become well-typed. Consider, for example, the term in Figure 7. The function $untilFalse$ takes as argument a computation that reads and writes high before returning a boolean, and runs that computation repeatedly until it returns false. Recursion is accomplished using backpatching: a store location with a dummy value is allocated and is bound to *wref*, recursive calls in the body of the loop dereference *wref* and run the contents. The recursive knot is tied by overwriting the contents of *wref* with the real loop body w.

Interestingly, although *untilFalse* takes a high-security computation as an argument, our type system is able to

give it the type $\bigcirc_{(\top,\top)}\mathsf{bool} \to \bigcirc_{(\bot,\top)}1$, that is its return type is a low-security computation. Intuitively, even if c is a high-security computation, $untilFalse\ c$ does not leak any information to low-security since any information gained from c's return value is used only within the loop.

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 [17, 16], "low" means below some fixed security level ζ, and "high" means not below ζ.

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.

4.1 Equivalence property

Formally, we define an equivalence property of computation states (and term and expressions) 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.

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:

$$\frac{\vdash H_1 : \Sigma_1 \quad \vdash H_2 : \Sigma_2 \quad \Sigma_1 \restriction U = \Sigma_2 \restriction U \quad \Sigma_1; \Sigma_2; \cdot \vdash H_1(\ell) \approx_\zeta H_2(\ell) : \Sigma_1(\ell) \text{ for } \ell \in U}{\vdash (H_1 : \Sigma_1) \approx_\zeta^U (H_2 : \Sigma_2)} \quad (33)$$

Where the notation $\Sigma \restriction X$ means Σ restricted to locations in the set X.

For a pair of computation states, only low security locations that are common to both computations are in view. Since allocation does not 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 (where $\downarrow(\zeta) = \{\ell \mid \mathsf{Level}(\ell) \sqsubseteq \zeta\}$ is the set of all low security locations) :

$$\frac{\vdash (H_1 : \Sigma_1) \approx_\zeta^{\mathrm{dom}(H_1) \cap \mathrm{dom}(H_2) \cap \downarrow(\zeta)} (H_2 : \Sigma_2) \quad \Sigma_1; \Sigma_2; \cdot \vdash E_1 \approx_\zeta E_2 \div_o A}{\vdash (H_1, \Sigma_1, E_1) \approx_\zeta (H_2, \Sigma_2, E_2) \div_o A} \quad (34)$$

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:

$$\frac{\Sigma_1; \Gamma \vdash V_1 : A \quad \Sigma_2; \Gamma \vdash V_2 : A \quad \vdash A \nearrow a \quad a \not\sqsubseteq \zeta}{\Sigma_1; \Sigma_2; \Gamma \vdash V_1 \approx_\zeta V_2 : A} \quad (35)$$

The remaining rules for term and expression equivalence are congruence rules that merely require corresponding sub-terms or sub-expressions to be equivalent. They are given in the extended paper.

4.2 Non-interference theorem

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

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, Σ_1, E_1) *and* (H_2, Σ_2, E_2) *such that* $\vdash (H_1 : \Sigma_1) \approx_\zeta^U (H_2 : \Sigma_2)$ *where* $U = \mathrm{dom}(\Sigma_1) \cap \mathrm{dom}(\Sigma_2) \cap {\downarrow}(\zeta)$*, and for* $i = 1, 2$*, there exist* C_i *and* $o_i = (r_i, w_i)$ *such that* $\Sigma_i; \cdot \vdash E_i \div_{o_i} C_i$ *and* $w_i \not\sqsubseteq \zeta$*. If* $(H_i, \Sigma_i, E_i) \to^* (H_i', \Sigma_i', E_i')$ *for* $i = 1, 2$ *then* $\vdash (H_1' : \Sigma_1') \approx_\zeta^{U'} (H_2' : \Sigma_2')$ *where* $U' = \mathrm{dom}(\Sigma_1') \cap \mathrm{dom}(\Sigma_2') \cap {\downarrow}(\zeta)$

The proof of this lemma appears in the extended version of the paper [2].

Lemma 4.2 (Hexagon Lemma). *For all* ζ*, if* $o = (r, w)$ *with* $r \sqsubseteq \zeta$*, and if* $\vdash S_1 \approx_\zeta S_2 \div_o C$ *and* $S_1 \to S_1'$*,* $S_2 \to S_2'$ *where* $S_1' \downarrow$ *and* $S_2' \downarrow$ *then there exist* S_1'', S_2'' *such that* $S_1' \to^* S_1''$*,* $S_2' \to^* S_2''$ *and* $\vdash S_1'' \approx_\zeta S_2'' \div_o C$

Proof. By Inversion on $\vdash S_1 \approx_\zeta S_2 \div_o C$, we get that each computation state S_i is a triple (H_i, Σ_i, E_i), and that the two stores and the two expressions are equivalent $\Sigma_1; \Sigma_2; \cdot \vdash E_1 \approx_\zeta E_2 \div_o C$. We prove the theorem by induction on this derivation. We consider one case below, the remaining cases are proved in the extended paper [2].

Case: $$\frac{\Sigma_1; \Sigma_2; \Gamma \vdash E_1 \approx_\zeta E_2 \div_{(r',w)} C \quad \vdash C \nearrow r'}{\Sigma_1; \Sigma_2; \Gamma \vdash E_1 \approx_\zeta E_2 \div_{(\bot,w)} C} \quad (36)$$

If $r' \sqsubseteq \zeta$, 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 $r' \not\sqsubseteq \zeta$, then since $r' \sqsubseteq w$, it follows that $w \not\sqsubseteq \zeta$ 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. □

Theorem 4.3 (Non-interference). *If* $\vdash H : \Sigma$ *and* $\Sigma; x : A \vdash E \div_{(r,w)} B$ *and if* $\Sigma; \Sigma; \cdot \vdash V_1 \approx_r V_2 : A$ *then if* $(H, \Sigma, E[V_1/x]) \to^* S_1$ *and* $(H, \Sigma, E[V_2/x]) \to^* S_2$ *and both* S_1, S_2 *are terminal, then* $\vdash S_1 \approx_r S_2 \div_{(r,w)} B$

Proof. By some easy structural properties, we can show that $\Sigma; \Sigma; \cdot \vdash E[V_1/x] \approx_r E[V_2/x] \div_{(r,w)} B$. By repeated application of the Hexagon Lemma, the two computations evaluate to equivalent terminal states. Since the operational semantics are deterministic (upto renaming of bound store locations), those terminal states are S_1 and S_2, respectively. □

5 Encoding a value-oriented language

A natural question is whether we sacrifice expressive power in comparison to value-oriented secure languages. 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. We consider the language $\lambda_{\mathrm{SEC}}^{\mathrm{REF}}$ (summarized in Figure 8) of Zdancewic [15] and show that it can be encoded into our language.

In an imperative setting, information gained via control-flow may leave an expression non-locally (*e.g.*, via a write to the store). As a result, it becomes necessary to track such *implicit flows* of information. Secure imperative

$$
\begin{array}{rcll}
t & \in & \textit{types} & ::= \quad 1 \mid \mathsf{bool} \mid s_1 \xrightarrow{\mathsf{pc}} s_2 \mid \mathsf{ref}\ s \\
s & \in & \textit{security types} & ::= \quad (t, a) \\
bv & \in & \textit{base values} & ::= \quad * \mid \mathsf{true} \mid \mathsf{false} \mid \ell \mid \lambda[\mathsf{pc}]x : s.e \\
e & \in & \textit{expressions} & ::= \quad x \mid bv_a \mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \mid e_1 e_2 \mid \mathsf{ref}\ (e : s)) \mid\ !e \mid e := e'
\end{array}
$$

Figure 8: $\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$ Syntax

languages use a so-called *program counter security level*, pc, as a lower bound on the information that a computation may gain via control flow. Consequently, the results and effects of each expression must be at least as secure as any information gained via control flow.

The typing rules for $\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$ are unsurprising for a value-oriented language (see the extended paper). The rule for lambda captures the program counter annotation in the arrow type, and an application expression releases the effects provided that they do not leak information through control flow or the return value.

$$
\frac{\Sigma; \Gamma, x : s[\mathsf{pc}] \vdash e : s'}{\Sigma; \Gamma \vdash \lambda[\mathsf{pc}]x : s.e\ : s \xrightarrow{\mathsf{pc}} s'}
\qquad
\frac{\Sigma; \Gamma[\mathsf{pc}] \vdash e_1 : (s' \xrightarrow{\mathsf{pc}'} s, a) \quad \Sigma; \Gamma[\mathsf{pc}] \vdash e_2 : s' \quad \mathsf{pc} \sqcup a \sqsubseteq \mathsf{pc}'}{\Sigma; \Gamma[\mathsf{pc}] \vdash e_1 e_2 : s \sqcup a}
$$

Encoding In order to emulate the sealing behavior of value-oriented languages in our store-oriented discipline, we embed source-language values of security type $s = (t, a)$ into read-only refs in our language $\overline{s} = \mathsf{refr}_a\, \overline{t}$.

In a $\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$ function of type $s \xrightarrow{\mathsf{pc}} s'$ the program counter annotation pc is a conservative approximation of the information gained by the body of the function. Therefore, values written by the body must have security level at least pc. Thus, the corresponding writes in the translation must have write level at least pc. Consequently, the corresponding translated type for a function is $\overline{s} \to \bigcirc_{(\bot,\mathsf{pc})}\, \overline{s'}$.

The encoding for $\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$ expressions is given by a pair of judgments $\Sigma; \Gamma \vdash bv : t \Rightarrow M$ and $\Sigma; \Gamma[\mathsf{pc}] \vdash e : s \Rightarrow E$, given in the extended paper. In [2] we show that the translation preserves typing, that is (extending $\bar{\cdot}$ pointwise to store types and to contexts) whenever $\Sigma; \Gamma[\mathsf{pc}] \vdash e : s \Rightarrow E$, it follows that $\overline{\Sigma}; \overline{\Gamma} \vdash E \div_{\bot,\mathsf{pc}} \overline{s}$.

Non-interference 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 an instance of non-interference for $\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$ is preserved by our translation.

Theorem 5.1 ($\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$ non-interference). *Suppose $\Sigma_0; x : (t, a)[b] \vdash f : (\mathsf{bool}, b) \Rightarrow F$ where $a \not\sqsubseteq b$, and suppose that H, Σ are such that $\Sigma \supseteq \overline{\Sigma_0}$, and $\vdash H : \Sigma$. If $\Sigma; \cdot \vdash \ell_i : \mathsf{refr}_a\, \overline{t}$ for $i = 1, 2$ and if there exist $H_1, H_2, \Sigma_1, \Sigma_2, V_1, V_2$ such that $(H', \Sigma', F[\ell_i/x]) \to^* (H_i, \Sigma_i, [V_i])$ for $i = 1, 2$, then $V_i = \ell'_i$ and $H_1(\ell'_1) = H_2(\ell'_2)$ as booleans.*

Proof. From the type-correctness of the translation, and since the argument locations ℓ_i are out of view, by the non-interference theorem we conclude that $\vdash (H_1, \Sigma_1, [V_1]) \approx_b (H_2, \Sigma_2, [V_2]) \div_{(b,b)} \mathsf{refr}_b\, \mathsf{bool}$.

By inversion and by a canonical forms lemma, each V_i must be some store location $\ell'_i \in \mathrm{dom}(\Sigma_i)$ and $\Sigma_1; \Sigma_2; \cdot \vdash \ell'_1 \approx_b \ell'_2 : \mathsf{refr}_b\, \mathsf{bool}$. Since each $\Sigma_i(\ell'_i)$ must be a subtype of $\mathsf{refr}_b\, \mathsf{bool}$, each $\mathsf{Level}(\ell'_i)$ must be below b, and the two locations must be in-view. Therefore, $\ell'_1 = \ell'_2$ and furthermore, the values in those locations must, in turn, be equivalent $\Sigma_1; \Sigma_2; \cdot \vdash H_1(\ell'_1) \approx_b H_2(\ell'_2) : \mathsf{bool}$. Since bool is informative at any security level, by inversion, it must be the case that $H_1(\ell'_1) = H_2(\ell'_2)$. □

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 [12] outlines the key ideas in the design of secure programming languages.

Our account is most related to the Dependency Core Calculus [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 *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.

When viewed through the lens of the encoding of (a pure subset of) $\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$, 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 $\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$ 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 (*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.

A further similarity exists between the *tampering levels* of Honda and Yoshida [5] and informativeness. They work in a concurrent setting of a typed π-calculus, and the tampering level of a process represents the least security level that may observe the effects of a process of a given type. They present a calculus in the style of [13] extended with local variables, reference types and higher-order procedures and a translation of it into their typed process calculus. Much of the complexity of their language stems from tracking the action set of a command, that is, the references (conflated with program variables) that a command may read or write. Our language may be seen as a restatement of their language in a more conventional monadic style. In the setting of [5], our upcall rule (exploiting the informativeness judgment) would correspond to leaving out the information that a command read from some variables from its action set whenever the command does not tamper below a certain security level.

Harrison *et al.* [3] observed that monads and monad transformers may be used to separate pieces of the state with different security levels, thus ensuring a kind of non-interference via properties of the state monad transformer. However their system does not statically rule out insecure flows when computations at different security levels are combined. Instead, the system dynamically prevents security leaks by channeling communication between computations at different security levels through a trusted kernel.

7 Conclusion

We give an account of secure information flow in the context of a higher-order language with mutable state. Moreover, motivated by a low-level store-oriented view of computation, we arrive at a view of security based on lax logic. 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. The question of how to account for additional effects requires further work. From the point of view of non-interference, effects introduce the possibility of different behavior from seemingly related expressions. We expect that by further refining the monadic type to restrict the behavior of related terms, we may be able to account for effects such as I/O or non-local control transfers.

Certain complications beyond those discussed in this paper remain in developing a typed assembly language that tracks information flow. One problem to be dealt with is the re-use of registers between low-security and high-security computations. Any mutation of a register by a high security computation could potentially be observed

once it returns to a low-security caller. As a result it is necessary to exploit informativeness to ensure that the contents of registers are not informative to the caller. We conjecture that informativeness in conjunction with linear continuations [17] will prove invaluable to the design of a secure TAL.

Our formulation of the monadic language is in the style of Pfenning and Davies [10]. 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.

Acknowledgments Thanks to Matthew Harren and Steve Zdancewic for their comments and suggestions on earlier drafts of this paper.

References

[1] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In *Twenty-Sixth ACM Symposium on Principles of Programming Languages*, pages 147–160, San Antonio, Texas, Jan. 1999.

[2] K. Crary, A. Kliger, and F. Pfenning. A monadic analysis of information flow security with mutable state. *Journal of Functional Programming*, 2004. To Appear. An earlier version is available as Carnegie Mellon University, School of Computer Science technical report CMU-CS-03-164.

[3] W. Harrison, M. Tullsen, and J. Hook. Domain separation by construction. In *Foundations of Computer Security Workshop(FCS'03)*, Ottowa, Canada, June 2003.

[4] N. Heintze and J. G. Riecke. The SLam calculus: Programming with secerecy and integrity. In *Twenty-Fifth ACM Symposium on Principles of Programming Languages*, pages 365 – 377, San Diego, California, Jan. 1998.

[5] K. Honda and N. Yoshida. A uniform type structure for secure information flow. In *Twenty-Ninth ACM Symposium on Principles of Programming Languages*, pages 81–92, Jan. 2002.

[6] E. Moggi. Computational lambda-calculus and monads. In *Fourth IEEE Symposium on Logic in Computer Science*, pages 14–23, 1989.

[7] E. Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[8] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999. An earlier version appeared in the 1998 Symposium on Principles of Programming Languages.

[9] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Twenty-Sixth ACM Symposium on Principles of Programming Languages*, pages 228–241, San Antonio, Texas, Jan. 1999.

[10] F. Pfenning and R. Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11(4):511–540, 2001.

[11] F. Pottier and V. Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems*, 25(1):117–158, Jan. 2003.

[12] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5 – 19, Jan. 2003. special issue on Formal Methods in Security.

[13] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. *Twenty-Fifth ACM Symposium on Principles of Programming Languages*, pages 355 – 364, 1998.

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

[15] S. Zdancewic. *Programming Languages for Information Security*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, New York, 2002.

[16] S. Zdancewic and A. C. Myers. Secure information flow and CPS. In *Tenth European Symposium on Programming*, volume 2028 of *Lecture Notes in Computer Science*, pages 46 – 61. Springer-Verlag, Apr. 2001.

[17] S. Zdancewic and A. C. Myers. Secure information flow via linear continuations. *Higher Order and Symbolic Computation*, 15(2-3):209–234, Sept. 2002.