The SLam Calculus: Programming with Secrecy and Integrity

Nevin Heintze
Bell Laboratories
Lucent Technologies
700 Mountain Avenue
Murray Hill, NJ 07974 USA
nch@bell-labs.com

Jon G. Riecke
Bell Laboratories
Lucent Technologies
700 Mountain Avenue
Murray Hill, NJ 07974 USA
riecke@bell-labs.com

Abstract

The **SLam calculus** is a typed λ-calculus that maintains *security information* as well as type information. The type system propagates security information for each object in four forms: the object's creators and readers, and the object's *indirect* creators and readers (*i.e.*, those agents who, through flow-of-control or the actions of other agents, can influence or be influenced by the content of the object). We prove that the type system prevents security violations and give some examples of its power.

1 Introduction

How do we build a system that manipulates and stores information whose secrecy and integrity must be preserved? The information might, for example, contain employee salaries, tax information or social security numbers, or it might involve data whose integrity is essential to global system security, such as the UNIX™ `/etc/passwd` file or a database of public keys.

One solution is to provide a secure persistent store that controls access to each location in the store, *e.g.*, it might maintain access control lists that specify who may read and write each location. This only addresses part of the problem: it does not trace the security of information through *computation*. For example, a privileged user can write a program that reads a secret location and copies its contents to a public location. Trust is central to the usability of this system: if one user shares secrets with others, he or she must trust their intentions. He or she must *also* trust their competence, since they may release secrets accidentally as a result of a programming error.

There is an alternative to a secure persistent store: we can associate security with data objects instead of locations, and track security through computation at runtime. To do this, each object must come equipped with security information that specifies access rights, *i.e.*, a **capability**. However, this is not enough. We must also track the flow of information as we build new objects from old. For example, if we have a secret string and concatenate it with another string, then we must also treat the new string as secret. Such a scheme has two problems. First, explicitly tracing security information through computation is very expensive. Second, the system must guarantee that security information is not forged.

We can address these two problems by annotating programs with information flow levels, and using a static analysis system that rejects programs that might leak information. For example, we can view a program as a black box so that its output is at least as secret as each of its inputs. Similarly, we can view program output as having no higher integrity than each of its inputs.

This approach of tracing information flow has been thoroughly explored in the security literature [3, 4, 6, 7, 8]. Unfortunately, in classic information flow systems, data quickly floats to the highest level of security. For example, consider a program that takes as input a string `x` representing a user id and another string `y` representing the user's password, and whose output is some object `z` built from `x` and `y`. Then the security level of the entire object `z` must be the security level appropriate for user passwords, even though the password information may be only a small component of `z`.

In this paper, we show that a programming language can provide finer grained control of security. The vehicle of study is a core functional programming language called the **Secure Lambda Calculus** (or SLam calculus). We focus on the role of *strong static typing*. We assume that all programs are compiled with a trusted compiler that enforces our type discipline, *i.e.*, there are no "backdoors" for inserting and executing unchecked code and there are no operations for direct access and modification of raw memory locations.

The types of the SLam calculus mingle security information with type information. This feature allows more accurate and flexible security information, in that we can attach different levels of security to the different components of an object. For instance, in the `/etc/passwd` list, password hashes might come with higher security than other components of the records. The security information takes four forms: readers, creators, indirect readers and indirect creators. Intuitively, an agent must be a reader of an object in order to inspect the object's contents. An agent is a creator of any object it constructs. An agent is an indirect reader of an object if it may be influenced by the object's contents. An agent is an indirect creator of an object if it may influence the object's construction. For example, consider the statement

```
if (x > 25) then y:=1 else y:=2
```

Here, partial information about `x` is available via variable `y`. If agent *A* can read `y` but not `x`, then *A* can still find out partial information about `x` and so *A* is an indirect reader of `x`. If the statement itself were executed by agent *B*, then *B* would require read access to `x`. *B* would also be a creator of `y`'s content. Moreover, if `x` was created by a third agent *C*, then *C* would be an indirect creator of `y`'s content.

Readers and indirect readers together specify an object's secrecy (who finds out about the object), whereas creators and in-

direct creators specify integrity (who is responsible for the object). Readers and creators together capture access control, while indirect readers and indirect creators capture information flow. In conjunction with higher-order functions and a rich underlying type structure (*e.g.*, records, sums), these four security forms provide a flexible basis for controlled sharing and distribution of information. For example, higher-order functions can be used to build up complex capabilities. In our system, the ability to read a function is the right to apply it. By specifying the set of readers and/or indirect readers of these functions, we can additionally restrict the capability so that it can only be shared by a specified group of agents.

Incorporating both access control and information flow in the type system adds flexibility. To illustrate the utility of having both, suppose we have just two security levels, `H` (high security) and `L` (low security), and a type called `users` that is a list containing strings whose direct readers are `H`, and whose indirect readers are `L`. If we ignore direct and indirect creators, and write direct readers before indirect readers, the type definition (in a ML-style syntax [11]) might look like

```
type users = (list (string,(H,L)),(L,L))
```

Now suppose we want to look up names in a value of type `users`, and return `true` if the name is in the list. We might write the code as follows:

```
fun lookup ([]:users) name =
    false:(bool,(L,L))
  | lookup ((x::tail):users) name =
    if x = name
       then true:(bool,(L,L))
       else lookup tail name
```

Our type system guarantees that only high-security agents can write the `lookup` function, or indeed any code that branches on the values of the strings in the list. Low-security agents can call the `lookup` function, but cannot get direct access to the strings held in the list. Information flows from the strings into the boolean output, but since we have labeled the *indirect* readers of the strings to be low security, the program is type correct.

Direct readers determine how much of an object is revealed to indirect readers. (Of course, no one can reveal information to agents who are not indirect readers.) At one extreme, a direct reader can reveal everything about an object to the indirect readers. If an agent is an indirect reader of an object but not a direct reader, then any information that agent finds out about that object must be via a direct reader.

To simplify the presentation, we describe the SLam calculus in stages. In Section 2, we define the purely functional core of the SLam calculus, restricting the security properties to direct and indirect readers. The operational semantics explicitly checks for security errors. We prove a type soundness result: well-typed programs never cause security errors, and hence the run-time security checks may be omitted. We also prove a "noninterference" theorem for indirect readers and creators. Borrowing ideas from Reynolds [18], we prove the noninterference theorem using denotational semantics and logical relations. This style of proof, while common in the languages literature, is novel to the security world. We believe the proof technique carries over to the extensions of the basic calculus. Sections 3 and 4 extend the core calculus with assignments (using an effects-style extension to the type system [20]), concurrency, and integrity. Sections 5 and 6 conclude the paper with a discussion of other work and limitations of the system.

2 The Purely Functional SLam Calculus

We illustrate the core ideas of our calculus using a language with functions, recursion, tuples, and sums, restricting the security properties to direct and indirect readers. We extend our treatment to a language with assignment and concurrency in Section 3, and to creators and indirect creators in Section 4.

2.1 Types and terms

The types of the SLam calculus essentially comprise those of a monomorphic type system—with products, sums, and functions—in which each type is annotated with security properties. (We could add other primitive types such as booleans, integers and strings—and will do so in examples—but the essential typing properties are already covered by sums and products.) Define **security properties** κ, **types** t, and **secure types** s by the grammar

$$\begin{array}{rcl} \kappa & ::= & (r, ir) \\ t & ::= & \mathtt{unit} \mid (s+s) \mid (s \times s) \mid (s \to s) \\ s & ::= & (t, \kappa) \end{array}$$

where r (readers) and ir (indirect readers) range over some collection S of basic security descriptions with ordering $\sqsubseteq$, and security properties (r, ir) satisfy $ir \sqsubseteq r$. In other words, r should be more restrictive (it represents a smaller set of agents) than ir. We assume $(S, \sqsubseteq)$ is a lattice (*i.e.*, a partially ordered set with meets, joins, a top element $\top$ and bottom element $\bot$). Higher in the lattice means "more secure"; $\top$ is the most secure element. Intuitively, each element of S represents a set of agents or users; for this reason we refer to elements of S as **security groups**. For example, a simple multilevel security system might have security descriptions L (low), M (medium) and H (high), with ordering $L \sqsubseteq M \sqsubseteq H$, where $L = \bot$ and $H = \top$. Alternatively, a UNIX-like security system could be built from a lattice S of sets of users ordered by superset.

For the purposes of presenting our static type system, we assume that S is static. This assumption is unrealistic in practice: new users can be added to a system, or users may have their privileges revoked. We briefly discuss this issue in Section 6.

The SLam calculus is a *call-by-value* language, and hence terms contain a set of values that represent terminated computations. The sets of **basic values** and **values** are defined by the grammar

$$\begin{array}{rcl} bv & ::= & () \mid (\mathtt{inj}_i\ v) \mid \langle v, v \rangle \mid (\lambda x : s.\, e) \\ v & ::= & bv_\kappa \end{array}$$

The security properties on values describe which agents may read the object, and which agents may indirectly depend on the value. The terms of the SLam calculus are given by the grammar

$$\begin{array}{rcl} e & ::= & x \mid v \mid (\mathtt{inj}_i\ e)_\kappa \mid \langle e, e \rangle_\kappa \mid (e\ e)_r \mid (\mathtt{proj}_i\ e)_r \mid \\ & & (\mathtt{protect}_{ir}\ e) \mid (\mu f : s.\, e) \mid \\ & & (\mathtt{case}\ e\ \mathtt{of}\ \mathtt{inj}_1(x).\, e \mid \mathtt{inj}_2(x).\, e)_r \end{array}$$

The term $(\mu f : s.\, e)$ defines a recursive function, and $(\mathtt{protect}_{ir}\ e)$ increases the security property of a term. Bound and free variables are defined in the usual way: variables may be bound by λ, μ, and `case`.

Notice that the destructors—application, projection, and case—come labeled with a security group r. This group represents the security group of the *programmer* of that code. The annotations appear on terms for entirely technical reasons: with the operational semantics given in the next section, terms with mixed annotations arise when code written by different programmers gets mixed together. For example, `root` can write a function

$$f = (\lambda x : (t, (\bot, \bot)).\, \textit{body with } \mathtt{root} \textit{ annotations})_{(\bot,\bot)},$$

that can be run by anyone; once called, the function's body accesses files and data structures as `root`, *i.e.*, has "setuid" behavior. Any user can write an application $(f\ v)_r$, because $\bot \sqsubseteq r$, but the body

can access data that r cannot access. Mixed annotations arise when such applications are reduced: when v is substituted for x in *body*, the result might contain r and `root` annotations.

A compiler for the SLam calculus must check that the annotations on programs are consistent with the programmer's security. For example, the compiler must prevent arbitrary users from writing programs with destructors annotated with `root`.

2.2 Operational semantics

The relation $e \to e'$ represents a single atomic action taken by an agent. The definition uses structured operational semantics [16] via **evaluation contexts** [9]. The set of evaluation contexts is given by

$$\begin{array}{rcl} E & ::= & [\cdot] \mid (E\ e)_r \mid (v\ E)_r \mid (\mathtt{proj}_i\ E)_r \mid (\mathtt{inj}_i\ E)_\kappa \mid \\ & & \langle E, e\rangle_\kappa \mid \langle v, E\rangle_\kappa \mid (\mathtt{protect}_{ir}\ E) \mid \\ & & (\mathtt{case}\ E\ \mathtt{of}\ \mathtt{inj}_1(x).\ e_1 \mid \mathtt{inj}_2(x).\ e_2)_r \end{array}$$

This defines a left-to-right, call-by-value, deterministic reduction strategy.

The basic rules for the operational semantics appear in Table 1. These rules reduce simple redexes. They lift to arbitrary terms via

$$\frac{e \to e'}{E[e] \to E[e']}$$

In the rules of Table 1, we use an operation for increasing the security properties on terms: given $\kappa = (r, ir)$, the expression $\kappa \bullet ir'$ is the security property $(r \sqcup ir', ir \sqcup ir')$. Abusing notation, we extend this operation to values: $bv_\kappa \bullet ir$ denotes the value $bv_{\kappa \bullet ir}$.

Note that the operational semantics is essentially *untyped*: the types on bound variables, upon which the type checking rules of the next section depend, are ignored during reduction. The security properties on values and destructors are, of course, checked during reduction; this corresponds to checking, for instance, that the argument to a projection is a pair and not, say, a function abstraction. To see how programs can get stuck at security errors, suppose the security lattice has two elements L and H, with $L \leq H$. Let

$$\begin{array}{rcl} \mathtt{bool} & = & (\mathtt{unit}, (L,L)) + (\mathtt{unit}, (L,L)) \\ \mathtt{true}_\kappa & = & (\mathtt{inj}_1\ ()_{(L,L)})_\kappa : (\mathtt{bool}, \kappa) \\ \mathtt{false}_\kappa & = & (\mathtt{inj}_2\ ()_{(L,L)})_\kappa : (\mathtt{bool}, \kappa) \end{array}$$

and let $(\mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3)_r$ abbreviate the term

$$(\mathtt{case}\ e_1\ \mathtt{of}\ \mathtt{inj}_1(x).\ e_2 \mid \mathtt{inj}_2(x).\ e_3)_r$$

where x is a variable not occurring in e_2 or e_3. Then the term

$$(\mathtt{if}\ \mathtt{true}_{(H,H)}\ \mathtt{then}\ \mathtt{true}_{(L,L)}\ \mathtt{else}\ \mathtt{false}_{(L,L)})_L$$

cannot be reduced: the programmer of the if-then-else code does not have the permissions to branch on the high-security boolean.

After a value has been destructed, the *indirect* readers of the value are used to increase the secrecy of the result (via `protect`). This tracks information flow from the destructed value to the result. For instance, if L is changed to H on the if-then-else statement in the above example, then the following reductions are permitted:

$$\begin{array}{l} (\mathtt{if}\ \mathtt{true}_{(H,H)}\ \mathtt{then}\ \mathtt{true}_{(L,L)}\ \mathtt{else}\ \mathtt{false}_{(L,L)})_H \\ \quad \to\ (\mathtt{protect}_H\ \mathtt{true}_{(L,L)}) \\ \quad \to\ \mathtt{true}_{(H,H)} \end{array}$$

This tracks the flow of information from the test to the result. We could also change the indirect reader annotation on the boolean test:

$$\begin{array}{l} (\mathtt{if}\ \mathtt{true}_{(H,L)}\ \mathtt{then}\ \mathtt{true}_{(L,L)}\ \mathtt{else}\ \mathtt{false}_{(L,L)})_H \\ \quad \to\ (\mathtt{protect}_L\ \mathtt{true}_{(L,L)}) \\ \quad \to\ \mathtt{true}_{(L,L)} \end{array}$$

In this case, only a high-security direct reader can test the boolean, but information about the test is revealed to low-security agents.

2.3 Type system

The type system of the SLam calculus appears in Tables 2 and 3. The system includes subtyping and the subsumption rule. The subtyping rules in Table 2 start from lifting the $\sqsubseteq$ relation on security groups to the $\leq$ relation on security properties. Define

$$(r, ir) \leq (r', ir') \quad \text{iff} \quad r \sqsubseteq r', ir \sqsubseteq ir'.$$

The subtyping rules formalize the idea that one may always increase the security property of a value.

The typing rules appear in Table 3. We use Γ to denote **typing contexts**, *i.e.*, finite partial maps from variables to secure types, and $\emptyset$ to denote the empty context. Abusing notation, we write $(t, \kappa) \bullet ir$ to denote the secure type $(t, \kappa \bullet ir)$. The rules for type-checking constructors are straightforward. For type-checking destructors, the rules guarantee that the destructor has the permission to destruct the value (apply a function, project from a pair, or branch on a sum). For instance, recall the term

$$(\mathtt{if}\ \mathtt{true}_{(H,H)}\ \mathtt{then}\ \mathtt{true}_{(L,L)}\ \mathtt{else}\ \mathtt{false}_{(L,L)})_L$$

from the previous section which could not be reduced by the operational semantics. This term cannot be typed with our rules. The type rules track information flow in a manner analogous to the operational semantics. For instance, the type of the term

$$(\mathtt{if}\ \mathtt{true}_{(H,H)}\ \mathtt{then}\ \mathtt{true}_{(L,L)}\ \mathtt{else}\ \mathtt{false}_{(L,L)})_H$$

from the previous section is $(\mathtt{bool}, (H,H))$.

The type system satisfies Subject Reduction and Progress (see Appendix for proofs).

Theorem 2.1 (Subject Reduction) *Suppose $\emptyset \vdash e : s$ and $e \to e'$. Then $\emptyset \vdash e' : s$.*

Theorem 2.2 (Progress) *Suppose $\emptyset \vdash e : s$ and e is not a value. Then there is a reduction $e \to e'$.*

These theorems show that, for well-typed, closed expressions, the security checks in the operational semantics are redundant.

It follows from subject reduction and progress that our type system enforces reader security. Consider an object O created with reader annotation r. We wish to establish that, for well-typed programs, the only agents who can read this object are those specified by r. Now, consider the operational semantics. By inspection, the operational semantics ensures two properties of reader annotations:

- Once an object is created, the only way for its reader annotation to change is via `protect`, but this only *increases* security, *i.e.*, sets more restrictive access to the object.
- If an agent attempts to read an object and the current reader annotation does not allow the agent to perform the read, then the operational semantics will "get stuck."

Hence, the operational semantics ensures that if an agent not specified by r (the initial annotation on O) ever attempts to read O, then reduction will get stuck. But subject reduction and progress show that well-typed programs never get stuck. It follows that our type system enforces reader security.

A similar kind of argument could be used to show that our type system enforces indirect reader security. Such an argument would rely upon the following claim: if an agent not specified by ir (the initial indirect reader annotation on O) ever attempts to find out information about O, then reduction will get stuck. While the operational semantics contains information-flow aspects, the claim is certainly not self-evident by inspection of the reduction rules (as was the case for (direct) readers).

Table 1: Operational Semantics.

$((\lambda x : s.\, e)_{r,ir}\ v)_{r'}$	$\rightarrow$	$(\mathtt{protect}_{ir}\ e[v/x])$	if $r \sqsubseteq r'$
$(\mathtt{proj}_i\ \langle v_1, v_2\rangle_{(r,ir)})_{r'}$	$\rightarrow$	$(\mathtt{protect}_{ir}\ v_i)$	if $r \sqsubseteq r'$
$(\mathtt{case}\ (\mathtt{inj}_i\ v)_{(r,ir)}\ \mathtt{of}\ \mathtt{inj}_1(x).\, e_1 \mid \mathtt{inj}_2(x).\, e_2)_{r'}$	$\rightarrow$	$(\mathtt{protect}_{ir}\ e_i[v/x])$	if $r \sqsubseteq r'$
$(\mu f : s.\, e)$	$\rightarrow$	$e[(\lambda x : s_1.\, ((\mu f : s.\, e)\ x)_r)_{(r,ir)}/f]$	if $s = (s_1 \rightarrow s_2, (r, ir))$
$(\mathtt{protect}_{ir}\ v)$	$\rightarrow$	$v \bullet ir$	

Table 2: Subtyping Rules for Pure Functional Language.

$$\frac{s_1 \le s_2 \quad s_2 \le s_3}{s_1 \le s_3} \qquad \frac{\kappa \le \kappa'}{(\mathtt{unit}, \kappa) \le (\mathtt{unit}, \kappa')}$$

$$\frac{\kappa \le \kappa' \quad s_1 \le s_1' \quad s_2 \le s_2'}{((s_1 + s_2), \kappa) \le ((s_1' + s_2'), \kappa')} \qquad \frac{\kappa \le \kappa' \quad s_1 \le s_1' \quad s_2 \le s_2'}{((s_1 \times s_2), \kappa) \le ((s_1' \times s_2'), \kappa')}$$

$$\frac{\kappa \le \kappa' \quad s_1' \le s_1 \quad s_2 \le s_2'}{((s_1 \rightarrow s_2), \kappa) \le ((s_1' \rightarrow s_2'), \kappa')}$$

Table 3: Typing Rules for Pure Functional Language.

[*Var*] $\Gamma, x : s \vdash x : s$

[*Unit*] $\Gamma \vdash ()_\kappa : (\mathtt{unit}, \kappa)$

[*Sub*] $$\frac{\Gamma \vdash e : s \quad s \le s'}{\Gamma \vdash e : s'}$$

[*Rec*] $$\frac{\Gamma, f : s \vdash e : s}{\Gamma \vdash (\mu f : s.\, e) : s}\ s \text{ is a function type}$$

[*Lam*] $$\frac{\Gamma, x : s_1 \vdash e : s_2}{\Gamma \vdash (\lambda x : s_1.\, e)_\kappa : (s_1 \rightarrow s_2, \kappa)}$$

[*App*] $$\frac{\Gamma \vdash e : (s_1 \rightarrow s_2, (r, ir)) \quad \Gamma \vdash e' : s_1}{\Gamma \vdash (e\ e')_{r'} : s_2 \bullet ir}\ r \sqsubseteq r'$$

[*Pair*] $$\frac{\Gamma \vdash e_1 : s_1 \quad \Gamma \vdash e_2 : s_2}{\Gamma \vdash \langle e_1, e_2\rangle_\kappa : (s_1 \times s_2, \kappa)}$$

[*Proj*] $$\frac{\Gamma \vdash e : (s_1 \times s_2, (r, ir))}{\Gamma \vdash (\mathtt{proj}_i\ e)_{r'} : s_i \bullet ir}\ r \sqsubseteq r'$$

[*Inj*] $$\frac{\Gamma \vdash e : s_i}{\Gamma \vdash (\mathtt{inj}_i\ e)_\kappa : (s_1 + s_2, \kappa)}$$

[*Protect*] $$\frac{\Gamma \vdash e : s}{\Gamma \vdash (\mathtt{protect}_{ir}\ e) : s \bullet ir}$$

[*Case*] $$\frac{\Gamma \vdash e : (s_1 + s_2, (r, ir)) \quad \Gamma, x : s_1 \vdash e_1 : s \quad \Gamma, x : s_2 \vdash e_2 : s}{\Gamma \vdash (\mathtt{case}\ e\ \mathtt{of}\ \mathtt{inj}_1(x).\, e_1 \mid \mathtt{inj}_2(x).\, e_2)_{r'} : s \bullet ir}\ r \sqsubseteq r'$$

Instead, we employ a more direct argument of indirect reader security. We show that if x is a high security variable (with respect to indirect readers) and e is a low security expression that contains x, then no matter what value we give to x, the resulting evaluation of e does not change (assuming it terminates). More generally, we show that if an expression e of low security has a high security subexpression, then we can arbitrarily change the high security subexpression without changing the value of e. This property, called "noninterference" in the security literature, states that high security subexpressions cannot interfere with low security contexts.

The statement of noninterference for the SLam calculus requires two technical conditions, both of which arise from the fact that the only base type in the language is the trivial type `unit`. First, since the language contains function expressions—which cannot be checked meaningfully for equality—the theorem only considers terms of **ground types**, *i.e.*, those types containing only `unit`, sums, and products. For instance, $(\texttt{bool},\kappa)$ is a ground type but $((\texttt{bool},\kappa)\to(\texttt{bool},\kappa),\kappa)$ is not. Second, values constructed from `unit`, sums, and products may contain components with different security annotations. For example, consider

$$((\texttt{bool},(H,H))\times(\texttt{bool},(H,H)),(L,L)).$$

Clearly it is acceptable for the high security components of a value to depend on a high security variable x; however it is not acceptable for the low security components to depend on high security x. To simplify the statement of non-interference, we further restrict attention to **transparent types** whose security properties do not increase as we descend into the type structure (a formal definition of transparent types appears in the Appendix). For instance, the above type is not transparent, but the type $((\texttt{bool},(L,L))+(\texttt{bool},(L,L)),(H,H))$ is transparent.

To formally state the noninterference property, we use contexts: $C[\cdot]$ denotes a context (expression with a hole in it); $C[e]$ denotes the expression obtained by filling context $C[\cdot]$ with expression e. We also define a special equivalence relation to factor out termination issues: $e\simeq e'$ if whenever both expressions halt at values, the values (when stripped of security information) are identical.

Theorem 2.3 (Noninterference) *Suppose* $\emptyset\vdash e,e':(t,(r,ir))$ *and* $\emptyset\vdash C[e]:(t',(r',ir'))$, t' *is a transparent ground type and* $ir\not\sqsubseteq ir'$. *Then* $C[e]\simeq C[e']$.

For simplicity, we have restricted this theorem to closed terms e; it can be generalized to open terms. The proof uses a denotational semantics of the language and a logical-relations-style argument; the Appendix gives a sketch. The proof is particularly simple, especially when compared with other proofs based on direct reasoning with the operational semantics (*cf.* [23]). The proof method can also be extended to more complicated type systems, including ones with recursive types (see [15] for the necessary foundations).

Notice that Noninterference Theorem requires *both* expressions to halt. This reveals a problem with the type system: it is vulnerable to timing attacks, *i.e.*, it does not protect the execution time of computations and termination/nontermination properties. For example, suppose we have an external observer who can simply test whether a program has terminated. Then we could write

```
let fun haltIfTrue x:(bool,(H,H)) =
        if x then ():(unit,(H,H))
             else haltIfTrue x
in  haltIfTrue secretBool;
    true:(bool,(L,L))
end
```

(taking some liberties with syntax) and the observer could discover partial information about the value of `secretBool`. The problem is worse if the observer has a timer. For example, consider the term

```
let fun i = if (i = 0) then ()
                       else f(i-1)
in  f secretValue:(int,(H,H))
end
```

in an extension of the SLam calculus with an integer base type. An observer can get some information about `secretValue` simply by timing the computation. If we add a `getTime` primitive to the calculus, this covert channel becomes observable and usable in the calculus; we can then write

```
let val t1 : (int,(L,L)) = getTime()
    val tmp = if secureBool:(bool,(H,H))
                 then longComp
                 else shortComp
    val t2 : (int,(L,L)) = getTime()
    val insecureBool : (bool,(L,L)) =
        ((t2 - t1) > timeForShortComp)
in  insecureBool
end
```

where `longComp` is some computation that takes longer than a computation `shortComp`. Here the contents of `secureBool` are leaked to `insecureBool`. The vulnerability here is dependent on the accuracy of the `getTime` primitive, latency issues and scheduling properties, and can be controlled by restricting access to primitives such as `getTime` and restricting the ability of external observers. Depending on the nature of the secret information, however, even a very low-rate covert channel can be disastrous, *e.g.*, a DES key can be leaked in about a minute over a 1 bit/second channel.

We remark that our type system could be modified to reduce exposure to timing attacks. The critical rule is the case rule: our current rule ensures that if the expression tested by the case statement has high security, then any value produced by either arm of the case statement must have high (indirect) security. To address timing attacks, we need to protect not just values returned by the arms of the case statement, but also values returned by *all future computation*. One approach is to introduce a notion of "the current security context of the computation", and propagate this in the rules of the type system. The type system described in the next section employs this idea. Another approach is to force both of a case statement to take the same time an space resources by adding timeout mechanisms and various padding operations. Still another approach may be found in [19, 21], which in our system would essentially correspond to restricting case statements so that the test expression of the case statement has low ($\bot$) security.

3 Assignment and Concurrency

The calculus in the previous section is single threaded and side-effect free. This is inadequate to model the behavior of a collection of agents that execute concurrently and interact, *e.g.*, via a shared store or file system. To model such a system, we extend the basic calculus with assignment (via ML-style reference cells), generalize evaluation to a multi-process setting, and add a "spawn" operation to create new processes.

The type system of a language with concurrency and side effects must be carefully designed: a naïve adaptation of the purely functional system is not sufficient. In fact, problems even arise with side effects alone. The problem lies in the sequencing of expressions (sequencing $(M;N)$ in the calculus can be programmed as $((\lambda d:s.N)\,M)_\top$, where d is a fresh "dummy" variable). Unlike the purely functional case, where the sequencing of two expressions communicates only termination behavior across the boundary, the sequencing of side-effecting expressions can communicate data. Consider, for instance, the term

Figure 1: Leakage of information in the concurrency setting.

```
P1: let fun loop1() =
              if secretBool
                   orelse (!killFlag)
                 then ()
                 else loop1()
        in  loop1();
            if not(!killFlag)
               then insecureBool := true
               else ();
            killFlag := true
        end
```

```
P2: let fun loop2() =
              if not(secretBool)
                   orelse (!killFlag)
                 then ()
                 else loop2()
        in  loop2();
            if not(!killFlag)
               then insecureBool := false
               else ();
            killFlag := true
        end
```

```
if secretBool:(bool,(H,H))
   then insecureBool:= true:(bool,(L,L));
        secretBool
   else insecureBool:= false:(bool,(L,L));
        secretBool
```

Assuming the `if` is of high security, the type of the expression is a high-security boolean. Nevertheless, the side effect has leaked out the value of `secretBool`. The operational semantics and the type system must therefore track such information flow from the destruction of values to side effects.

Concurrency raises other, more difficult issues. It is well documented that the notion of noninterference itself is problematic in a concurrent setting (see, *e.g.*, [10]). To recap the problem, consider a system with three agents *A*, *B* and *C*, and suppose that there is some variable `x` that contains information that should be kept secret from agent *C*. The first agent, *A*, generates a random number, and puts it into a variable `tmp` that everyone can read. Agent *B* waits for some time and then copies the contents of `x` into `tmp`. Agent *C* reads `tmp` and immediately terminates. Although *C* cannot tell that it has captured the contents of `x` or some random value, information about `x` has been leaked. If this interaction is repeated, then *C* can determine `x` to a high degree of certainty. However, if we use a standard set-of-behaviors semantics for this system, then we find that the set of behaviors of *C* (in particular, the values *C* generates) is independent of the initial value of `x`. Hence, if a standard concurrency semantics is employed, we might conclude that the system is secure.

A second issue is closely related to the issue of timing attacks discussed in the previous section. In that section, we justified ignoring timing attacks by viewing them as external to the system (we excluded `getTime` from that calculus). In a concurrent setting, however, ignoring timing attacks can cause serious security holes, because, in a calculus with concurrent communicating processes, the ability to "time" other processes is implicit. To see this, it is helpful to consider a version of the main example from [19]. In the example, we run complementary versions of the looping example from Section 2.3 in parallel, one of which loops if a secret is true, and the other loops if the secret is false. If the first loop terminates, the process leaks `true`, and if the second terminates, the process leaks `false`. We can even force the entire system to terminate by killing both processes if either loop terminates. Specifically, we can write processes P1 and P2 in Figure 1 (using a syntax similar to that of Standard ML [11]), where the global variable `killFlag` is initially `false`. The value `insecureBool` is ready when `killFlag` becomes true. To summarize, timing-style attacks in a concurrent setting are part of the system: it does not seem reasonable to analyze the security of a concurrent setting without taking them into account.[1]

[1]In fact, while the attack detailed above is clearly a generalization of the notion of timing attack in the sequential setting, it is not clear the phrase "timing attack" is appropriate.

3.1 Types and terms

To construct the enhanced language, we first extend the definition of basic values *bv* and expressions *e* by

$$\begin{array}{rcl} bv & ::= & l^s \mid \ldots \\ e & ::= & (\mathtt{ref}_s\, e)_\kappa \mid (e := e)_r \mid (!e)_r \mid (\mathtt{spawn}_{ir}\, e)_\kappa \mid \ldots \end{array}$$

where l^s is a location (we assume an infinite sequence of locations at each type *s*; whenever a new location is needed, we use the next available location in the sequence). We modify the definition of types *t* to include reference types and change arrow types so that they carry a latent "effect" *ir*, representing a lower bound on the security of cells that may be written when the function is executed:

$$\begin{array}{rcl} \kappa & ::= & (r, ir) \\ t & ::= & \mathtt{unit} \mid (s+s) \mid (s \times s) \mid (s \xrightarrow{ir} s) \mid (\mathtt{ref}\, s) \end{array}$$

3.2 Operational semantics

The operational semantics of the enhanced language uses the same basic style as for the purely functional case, with rewrite rules for redexes and evaluation contexts to denote the position of redexes. The notion of evaluation contexts is a simple extension of the previous definition:

$$E \quad ::= \quad (\mathtt{ref}_s\, E)_\kappa \mid (E := e)_r \mid (v := E)_r \mid (!E)_r \mid \ldots$$

The semantics must now keep track of three new pieces of information. First, since there are multiple processes and not a single term, we must keep track of the current form of each process. Thus, the configuration of the system contains a list of processes. Second, each process must keep track of the security levels of previously destructed values, so that side effects can be appropriately coerced. For instance, in reducing the term

```
if secretBool:(bool,(H,H))
   then insecureBool:= true:(bool,(L,L));
        secretBool
   else insecureBool:= false:(bool,(L,L));
        secretBool
```

the rules will track the information flow *H* from `secretBool` to the values stored in the reference cell `insecureBool`. Third, the semantics must keep track of the values of the locations via a **state**, a finite partial function from typed locations l^s to values.

The basic rewrite rules for the enhanced language are found in Table 4. Specifically, the rules defines a reduction relation

$$((ir_1, e_1), \ldots, (ir_n, e_n); \sigma) \Rightarrow ((ir'_1, e'_1), \ldots, (ir'_{n+k}, e'_{n+k}); \sigma')$$

Table 4: Operational Semantics for Effects.

$$
\begin{array}{lcll}
(ir', ((\lambda x : s.\, e)_{r,ir}\ v)_{r'}) & \to & (ir \sqcup ir', (\mathtt{protect}_{ir}\ e[v/x])) & \text{if } r \sqsubseteq r' \\
(ir', (\mathtt{proj}_i\ \langle v_1, v_2\rangle_{r,ir})_{r'}) & \to & (ir \sqcup ir', (\mathtt{protect}_{ir}\ v_i)) & \text{if } r \sqsubseteq r' \\
(ir', (\mathtt{case}\ (\mathtt{inj}_i\ v)_{r,ir}\ \mathtt{of}\ \mathtt{inj}_1(x).\ e_1 \mid \mathtt{inj}_2(x).\ e_2)_{r'}) & \to & (ir \sqcup ir', (\mathtt{protect}_{ir}\ e_i[v/x])) & \text{if } r \sqsubseteq r' \\
(ir', (\mathtt{protect}_{ir}\ v)) & \to & (ir', v \bullet ir) &
\end{array}
$$

$$
\frac{(ir, e) \to (ir', e')}{(\ldots, (ir, E[e]), \ldots; \sigma) \Rightarrow (\ldots, (ir', E[e']), \ldots; \sigma)}
$$

$$
\begin{array}{lcll}
(\ldots, (ir', E[(\mathtt{spawn}_{ir}\ e)_\kappa]), \ldots; \sigma) & \Rightarrow & (\ldots, (ir \sqcup ir', e), (ir', E[()_\kappa]), \ldots; \sigma) & \\
(\ldots, (ir', E[(\mathtt{ref}_s\ v)_\kappa]), \ldots; \sigma) & \Rightarrow & (\ldots, (ir', E[l^s_\kappa]), \ldots; \sigma[l^s \mapsto v \bullet ir']) & \text{if } l^s \notin dom(\sigma) \\
(\ldots, (ir', E[(l^s_{(r,ir)} := v)_{r'}]), \ldots; \sigma) & \Rightarrow & (\ldots, (ir', E[v]), \ldots; \sigma[l^s \mapsto v \bullet ir']) & \text{if } r \sqsubseteq r' \\
(\ldots, (ir', E[(!l^s_{(r,ir)})_{r'}]), \ldots; \sigma) & \Rightarrow & (\ldots, (ir', E[\sigma(l^s) \bullet ir]), \ldots; \sigma) & \text{if } r \sqsubseteq r'
\end{array}
$$

Table 5: Typing Rules for Effects.

$$
\begin{array}{ll}
[Sub] & \dfrac{\Gamma \vdash_{ir} e : s \quad s \le s'}{\Gamma \vdash_{ir} e : s'} \\[2ex]
[Var] & \Gamma, x : s \vdash_{ir} x : s \\[2ex]
[Unit] & \Gamma \vdash_{ir} ()_\kappa : (\mathtt{unit}, \kappa) \\[2ex]
[Lam] & \dfrac{\Gamma, x : s_1 \vdash_{ir'} e : s_2}{\Gamma \vdash_{ir} (\lambda x : s_1.\, e)_\kappa : (s_1 \xrightarrow{ir'} s_2, \kappa)} \\[2ex]
[Pair] & \dfrac{\Gamma \vdash_{ir} e_1 : s_1 \quad \Gamma \vdash_{ir} e_2 : s_2}{\Gamma \vdash_{ir} \langle e_1, e_2\rangle_\kappa : (s_1 \times s_2, \kappa)} \\[2ex]
[Inj] & \dfrac{\Gamma \vdash_{ir} e : s_i}{\Gamma \vdash_{ir} (\mathtt{inj}_i e)_\kappa : (s_1 + s_2, \kappa)} \\[2ex]
[App] & \dfrac{\Gamma \vdash_{ir'} e : (s_1 \xrightarrow{ir'} s_2, (r, ir)) \quad \Gamma \vdash_{ir'} e' : s_1}{\Gamma \vdash_{ir'} (e\ e')_{r'} : s_2 \bullet ir}\ r \sqsubseteq r',\ ir \sqsubseteq ir' \\[2ex]
[Proj] & \dfrac{\Gamma \vdash_{ir'} e : (s_1 \times s_2, (r, ir))}{\Gamma \vdash_{ir'} (\mathtt{proj}_i\ e)_{r'} : s_i \bullet ir}\ r \sqsubseteq r',\ ir \sqsubseteq ir' \\[2ex]
[Case] & \dfrac{\Gamma \vdash_{ir'} e : (s_1 + s_2, (r, ir)) \quad \Gamma, x : s_1 \vdash_{ir'} e_1 : s \quad \Gamma, x : s_2 \vdash_{ir'} e_2 : s}{\Gamma \vdash_{ir'} (\mathtt{case}\ e\ \mathtt{of}\ \mathtt{inj}_1(x).\ e_1 \mid \mathtt{inj}_2(x).\ e_2)_{r'} : s \bullet ir}\ r \sqsubseteq r',\ ir \sqsubseteq ir' \\[2ex]
[Protect] & \dfrac{\Gamma \vdash_{ir'} e : s}{\Gamma \vdash_{ir'} (\mathtt{protect}_{ir}\ e) : s \bullet ir}\ r \sqsubseteq r',\ ir \sqsubseteq ir' \\[2ex]
[Spawn] & \dfrac{\Gamma \vdash_{ir} e : s}{\Gamma \vdash_{ir'} (\mathtt{spawn}_{ir}\ e)_\kappa : (\mathtt{unit}, \kappa)}\ ir' \sqsubseteq ir \\[2ex]
[Loc] & \Gamma \vdash_{ir'} l^s_\kappa : (\mathtt{ref}\ s, \kappa) \\[2ex]
[Ref] & \dfrac{\Gamma \vdash_{ir'} e : s}{\Gamma \vdash_{ir'} (\mathtt{ref}_s\ e)_\kappa : (\mathtt{ref}\ s, \kappa)}\ (s \bullet ir') = s \\[2ex]
[Assign] & \dfrac{\Gamma \vdash_{ir'} e_1 : (\mathtt{ref}\ s, (r, ir)) \quad \Gamma \vdash_{ir'} e_2 : s}{\Gamma \vdash_{ir'} (e_1 := e_2)_{r'} : s}\ r \sqsubseteq r',\ (s \bullet ir') = s \\[2ex]
[Deref] & \dfrac{\Gamma \vdash_{ir'} e : (\mathtt{ref}\ s, (r, ir))}{\Gamma \vdash_{ir'} (!e)_{r'} : s \bullet ir}\ r \sqsubseteq r'
\end{array}
$$

where σ, σ' are states. The first four rules define the relation $\rightarrow$, reductions for purely functional redexes. These reductions are lifted to general configurations of the form $((ir_1, e_1), \ldots, (ir_n, e_n); \sigma)$ with evaluation contexts. The rules for assignments, dereferencing, and process creation are non-local rules that are specified directly on configurations.

3.3 Type system

Subtyping in the system with effects is exactly the same as before, except that the rule for function types now becomes

$$\frac{\kappa \leq \kappa' \quad s_1' \leq s_1 \quad s_2 \leq s_2'}{((s_1 \xrightarrow{ir} s_2), \kappa) \leq ((s_1' \xrightarrow{ir} s_2'), \kappa')}$$

and the rule for reference types is

$$\frac{\kappa \leq \kappa'}{(\texttt{ref}\ s, \kappa) \leq (\texttt{ref}\ s, \kappa')}$$

Note that subtyping on reference types only affects top-level security properties, *i.e.*, subtyping is *invariant* on reference types. This restriction follows the standard subtyping rule for references [5, 23].

Table 5 presents the typing rules for the extended calculus. This type system is essentially the previous system with an effect system layered over the top of it in the style of [20]. This effect system tracks potential information leakage/dependency that may be introduced by reference cells. Each context carries with it a security group *ir* that is a lower bound on the security of the reference cells that may be written in that context; as expected, this security group is carried over onto arrow types. Only `spawn` terms may change the security context, and only then by increasing the context.

Analogs of the Subject Reduction Theorem 2.1 and Progress Theorem 2.2 can be established for this system; the proofs are quite similar to the proofs in the Appendix.

Theorem 3.1 (Subject Reduction) *Suppose σ is well-typed. Suppose that for all i, $\emptyset \vdash_{ir_i} e_i : s_i$, and*

$$((ir_1, e_1), \ldots, (ir_n, e_n); \sigma) \Rightarrow ((ir_1', e_1'), \ldots, (ir_{n+k}', e_{n+k}'); \sigma').$$

Then

- *σ' is well-typed;*
- *For all $1 \leq i \leq n$, $\emptyset \vdash_{ir_i'} e_i' : s_i$; and*
- *For all $i \geq (n+1)$, there is an s_i such that $\emptyset \vdash_{ir_i'} e_i' : s_i$.*

Theorem 3.2 (Progress) *Suppose σ is well-typed. Suppose that for all i, $\emptyset \vdash_{ir_i} e_i : s_i$, and $((ir_1, e_1), \ldots, (ir_n, e_n); \sigma)$. If e_i is not a value, then there is a reduction*

$$((ir_1, e_1), \ldots, (ir_n, e_n); \sigma) \Rightarrow ((ir_1', e_1'), \ldots, (ir_{n+k}', e_{n+k}'); \sigma').$$

We have not proved a noninterference theorem for the concurrent setting because, as mentioned earlier, the notion is unclear in the concurrency setting. Noninterference should, at the very least, guarantee that the set of possible values of low-security objects is independent of the initial values of high-security objects.

We might, however, expect more from a proof of noninterference. Ideally, we would like to rule out timing attacks: low-security observers should not be able to get and use information about the values of high-security objects through watching termination/nontermination or the timing behavior of programs. This raises an interesting issue: what is the notion of an "observer" in this setting? Suppose we consider observers to be *internal* to the system—that is, observers are simply processes running in parallel. If observers are internal, we conjecture that our system is secure. Intuitively, processes may only communicate through reference cells, and all writes to reference cells are protected by the current security context (see the operational rule for := in Table 4 and the corresponding type rule [*Assign*] in Table 5). For instance, the example of Figure 1 does not type check in our calculus. A simpler example can be made by modifying the looping example from the previous section:

```
let fun haltIfTrue x:(bool,(H,H)) =
        if x then ():(unit,(H,H))
             else haltIfTrue x
in  haltIfTrue secretBool;
    y := true
end
```

For this term to type check, we must use the `H` context (*i.e.*, with `H` on the $\vdash$), and so the type of *y* must be `(ref(bool,(H,H)),`κ`)`. Only high-security agents may therefore branch on the value held in *y*.

For *external* observers, *i.e.*, those that can observe the final answers of processes or simply the termination behavior of those processes, our system is not secure. For instance, the term

```
let fun haltIfTrue x:(bool,(H,H)) =
        if x then ():(unit,(H,H))
             else haltIfTrue x
in  haltIfTrue secretBool;
    true:(bool,(L,L))
end
```

from the last section still type checks in the `H` context. From the point of view of internal observers, this expression is secure, since the value `true:(bool,(L,L))` is not communicated to any other process in the system. From the point of view of external observers, however, the expression is not secure: we should make the type of the expression be `(bool,(H,H))`. We conjecture that the type system can be easily modified to address this issue.

The distinction between external and internal observers is familiar in the security world. The Spi calculus of [2], for instance, assumes that observers of protocols are those processes that can be programmed in the Spi calculus itself. Of course, this limits what observers can do, but it also makes precise the underlying assumptions of the model.

4 Integrity

We now sketch how to add integrity to the basic calculus of Section 2 and the extended calculus of Section 3, using the concepts of creators and indirect creators. Recall that *creators* track the agents that directly built the value, whereas *indirect creators* track the agents that may have influence over the eventual choice of a value.

Creators and indirect creators are drawn from the same underlying hierarchy of security groups as readers and indirect readers. High integrity is modeled by points near the top of the hierarchy, low integrity by points near the bottom. But there is a twist with respect to subtyping. Recall that for readers, one may always *restrict* access to a value, *e.g.*, change the reader annotation to a higher security group. For creators, it works just the opposite way: one may always *weaken* the integrity of a value, *e.g.*, change the creator annotation to a *lower* security group.

Security properties now incorporate creator and indirect creator information:

$$\kappa \quad ::= \quad (r, ir, c, ic).$$

The variables r, ir, c and ic range over security groups; we maintain the invariant that $ic \sqsubseteq c$. Subsumption for κ becomes

$$(r, ir, c, ic) \leq (r', ir', c', ic')$$
$$\text{iff } r \sqsubseteq r',\ ir \sqsubseteq ir',\ c' \sqsubseteq c, \text{ and } ic' \sqsubseteq ic$$

which formalizes the intuition that one may always weaken the integrity of a value. The definition of $\bullet$ must also be extended to the new context:

$$(r, ir, c, ic) \bullet (ir', ic') = (r \sqcup ir', ir \sqcup ir', c \sqcap ic', ic \sqcap ic').$$

We extend the operation $\bullet$ as well to values and types in the straightforward manner.

The operational semantics must now track indirect creators. For example, the rule for `case` becomes

$$(\texttt{case}\,(\texttt{inj}_j\, v)_{r,ir,c,ic}\ \texttt{of}\ \texttt{inj}_1(x).\, e_1 \mid \texttt{inj}_2(x).\, e_2)_{r'} \to (\texttt{protect}_{ir,ic\sqcap r'}\ e_j[v/x]) \quad \text{if } r \sqsubseteq r'$$

Note that the `protect` operation must take into account indirect creators. The rule registers the reader r' of the injected value as an indirect creator of the result of the computation. Typing rules that involve the $\bullet$ operation must be modified. For example, the `case` rule becomes

$$\frac{\Gamma \vdash e : (s_1 + s_2, (r, ir, c, ic)) \qquad r \sqsubseteq r' \qquad \Gamma, x : s_1 \vdash e_1 : s \qquad \Gamma, x : s_2 \vdash e_2 : s}{\Gamma \vdash (\texttt{case}\ e\ \texttt{of}\ \texttt{inj}_1(x).\, e_1 \mid \texttt{inj}_2(x).\, e_2)_{r'} : s \bullet (ir, ic \sqcap r')}$$

We have proven Subject Reduction and Progress Theorems analogous to Theorems 2.1 and 2.2 for this system. We can also prove a security result for indirect creators that is analogous to Theorem 2.3:

Theorem 4.1 (Noninterference) *Suppose* $\emptyset \vdash e, e' : (t, (r, ir, c, ic))$. *Suppose also that* $\emptyset \vdash C[e] : (t', (r', ir', c', ic'))$, t' *is a transparent ground type and* $ir \not\sqsubseteq ir'$ *and* $ic' \not\sqsubseteq ic$. *Then* $C[e] \simeq C[e']$.

Intuitively, if the indirect creators of the subexpression e do not include that of the entire computation, then e cannot influence the result of the computation. The proofs of these results use the techniques established in the Appendix.

Creators and indirect creators can also be added to the calculus of Section 3. Recall that in the case of readers, the type system must guarantee that information does not leak out via side effects. A similar property must be guaranteed in the case of creators: we must make sure that indirect creators of the computation are carried over onto the values written in reference cells. Therefore, judgements $\Gamma \vdash_{ir} e : s$ must be changed to $\Gamma \vdash_{ir,ic} e : s$, where ic is a lower bound on the integrity of values that may be written to reference cells in the evaluation of e. As before, indirect information is placed over $\to$ to represent the latent effects of a computation. The type-checking rules for abstraction and application thus become

$$[Lam] \quad \frac{\Gamma, x : s_1 \vdash_{ir',ic'} e : s_2}{\Gamma \vdash_{ir,ic} (\lambda x : s_1.\, e)_\kappa : (s_1 \xrightarrow{ir',ic'} s_2, \kappa)}$$

$$[App] \quad \frac{\Gamma \vdash_{ir',ic'} e : (s_1 \xrightarrow{ir',ic'} s_2, (r, ir, c, ic)) \qquad \Gamma \vdash_{ir',ic'} e' : s_1}{\Gamma \vdash_{ir',ic'} (e\, e')_{r'} : s_2 \bullet (ir, ic \sqcap r')} \quad \begin{array}{l} r \sqsubseteq r' \\ ir \sqsubseteq ir' \\ ic' \sqsubseteq ic \end{array}$$

We have proven Subject Reduction and Progress Theorems for this system; the proofs follow the structure of the proofs in the Appendix for the pure case.

5 Related Work

Our work is certainly not the first to use a static system or programming language framework for security. Other work addresses pure and modified information-flow models, and type systems for other security problems.

Work on information flow reaches back to Denning's work [6, 7] in the mid 1970s, and has been implemented in a variety of contexts (*e.g.*, the interpreter for Perl 5.0 can be put into a special dynamic "taint checking" mode which tracks information flow and rejects programs that may reveal secret information.) More modern treatments reformulate Denning's system in a static type system. For instance, Volpano, Smith and Irvine [23] provide one such reformulation for the simple language of "while" programs, and prove a version of the noninterference theorem. More recent work of Volpano and Smith [22] extends the language with first-order procedures; the types are similar to the types in Section 3, where annotations for the latent effect of a function are part of the type (written above the $\to$ in our types). Volpano and Smith also consider covert flows in the language of "while" programs with exceptions [21], and explore the example of Figure 1 in the language with nondeterminism [19]. Covert flows and the example of Figure 1 are eliminated by the same restriction: the tests and bodies of "while" loops must be of lowest security. In both cases, they prove the system is correct with a modification of the noninterference theorem: if a low-security program halts, then starting it in any state with different values for the high-security variables also halts, and produces the same low-security outputs. Our noninterference theorems are weaker: they say only that if *both* programs halt, they produce the same low-security outputs. On the other hand, the practicality of the restrictions on "while" loops is unclear.

Others have considered means to alleviate the problems with information flow. Myers and Liskov [13], for instance, describe a system which tracks information flow, but where agents may "declassify" information that they own. Ownership corresponds quite closely to our notion of direct reader: only direct readers may reveal information. In fact, one may build "declassify" operations from the primitives of the SLam calculus relatively straightforwardly by induction on types. Nothing seems to be known, however, about the formal properties of the system: rough ideas of the typing rules are given, but no correctness theorems are stated or proved.

There are other type systems for security. For example, Abadi's type system for the Spi calculus is used to reason about protocols [1, 2]. Type systems have been also used for the related problem of reasoning about trustworthiness of data. For instance, [14] introduces a calculus in which one can explicitly annotate expressions as trusted or distrusted and check their trust/distrust status; this system enforces consistent use of these annotations, although one can freely coerce from trusted to distrusted and vice-versa. Concurrency issues were first addressed by [4], although there appear to be some difficulties with that approach—see [23].

There are two main novelties of our work. First, we consider the purely functional language in isolation, and then extend it to side effects and concurrency. The type rules are less restrictive in the purely functional case than in the full language, so programming in the pure subset can lead to cleaner programs. Second, the type system combines both access control and information flow in a language with higher-order functions and data structures, and studies the system formally. These elements are essential for a development of practical languages that provide mechanisms for security, and introduce a number of new technical issues that have not been previously addressed.

6 Discussion

We view the SLam calculus as a first step towards providing a language basis for secure systems programming. It deals with the essence of computing with secure information, but a number of important issues remain. First, as with many other security systems, the SLam calculus relies on a TCB (trusted computer base): trusted type-checking, compilation, and runtime infrastructure. A failure in any of these components potentially breaks the entire security system. It would be possible to factor out some of the critical components by moving to a bytecode/bytecode-verifier organization (*à la* Java), although the benefits of doing so are unclear.

Second, the type system we have presented is monomorphic. Clearly this is too restrictive: we need to be able to write code that behaves uniformly over a variety of security groups (*e.g.*, in writing a generic string editing/searching package). We are currently investigating two approaches to this problem: parametric security types and a notion of "type dynamic" for security types. The former involves bounded quantification, and it is not clear we can compute concise, intuitive representations of types; the latter involves runtime overheads.

Third, our type system is static, but security changes dynamically. For instance, in a file system, the files that one can read today will probably be different from those one can read tomorrow. How can we accommodate new files, new objects, new cells, new agents, and changing security groups? We plan to address these issues using a dynamically typed object manager. The basic idea is that access to shared objects is via the object manager; although each program is statically typed, a program's interface to the object manager is via dynamic types (at runtime, a dynamically typed object returned from the object manager must be unpacked and its security properties checked before the raw object it contains is passed to the internals of the program).

Fourth, any practical language based on the SLam calculus must provide ways to reduce the amount of type information that must be specified by a programmer; the core SLam calculus is an explicitly typed calculus. Can we perform effective type reconstruction? What kinds of language support should we provide? For example, it would be useful to introduce a statically scoped construct that defines a default security group for all objects created in its scope, *i.e.*, like UNIX's `umask`.

We are investigating these and other issues in the context of an implementation of our type system for Java. While many of the appropriate typing rules for Java can be adapted easily from the SLam calculus, some new issues arise from exceptions, `break`, `continue`, `return`, and `instanceOf`. The implementation is joint work with Philip Wickline.

It would also be interesting to investigate other static analysis techniques to determine information flow. For instance, in the language with side effects, the only time the security context may change is in a `spawn` expression. This may be too restrictive in practice: statements that are executed before, say, a secure `case` statement need not restrict information flow so much. Abstract interpretation or set-based analyses might prove helpful here.

Acknowledgements: We thank Kathleen Fisher, Geoffrey Smith, Ramesh Subrahmanyam, Dennis Volpano, Philip Wickline, and the anonymous referees for helpful comments.

References

[1] M. Abadi. Secrecy by typing in security protocols. In *Theoretical Aspects of Computer Software: Third International Symposium*, volume 1281 of *Lect. Notes in Computer Sci.* Springer-Verlag, 1997.

[2] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *Proceedings of the 4th ACM Conference on Computer and Communications Security*, pages 36–47, 1997.

[3] G. Andrews and R. Reitman. An axiomatic approach to information flow in programs. *ACM Trans. Programming Languages and Systems*, 2(1):56–76, 1980.

[4] J. Banâtre, C. Bryce, and D. L. Metáyer. Compile-time detection of information flow in sequential programs. In European Symposium on Research in Computer Security, number 875 in Lect. Notes in Computer Sci., pages 55–73. Springer-Verlag, 1994.

[5] L. Cardelli. Amber. In *Combinators and functional programming languages, Proceedings of the 13th Summer School of the LITP*, volume 242 of *Lect. Notes in Computer Sci.* Springer-Verlag, 1986.

[6] D. Denning. *Secure Information Flow in Computer Systems*. PhD thesis, Purdue University, 1975.

[7] D. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–242, 1976.

[8] D. Denning and P. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977.

[9] M. Felleisen. The theory and practice of first-class prompts. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 180–190. ACM, 1988.

[10] D. McCullough. Noninterference and the composability of security properties. In *1988 IEEE Symposium on Security and Privacy*, pages 177–186, 1988.

[11] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[12] J. C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.

[13] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*. ACM Press, 1997.

[14] J. Palsberg and P. Ørbæk. Trust in the λ-calculus. In *Proceedings of the 1995 Static Analysis Symposium*, number 983 in Lect. Notes in Computer Sci. Springer-Verlag, 1995.

[15] A. M. Pitts. Relational properties of domains. *Information and Computation*, 127:66–90, 1996.

[16] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus Univ., Computer Science Dept., Denmark, 1981.

[17] G. D. Plotkin. (Towards a) logic for computable functions. Unpublished manuscript, CSLI Summer School Notes, 1985.

[18] J. C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pages 513–523. North Holland, Amsterdam, 1983.

[19] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Conference Record of the Twenty-Fifth Annual ACM Symposium on Principles of Programming Languages*. ACM, 1998.

[20] J.-P. Talpin and P. Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2:245–271, 1992.

[21] D. Volpano and G. Smith. Eliminating covert flows with minimum typings. In *Proceedings of the Tenth IEEE Computer Security Foundations Workshop*. IEEE Computer Society, 1997.

[22] D. Volpano and G. Smith. A type-based approach to program security. In *TAPSOFT'97: Theory and Practice of Software Development*, volume 1214 of *Lect. Notes in Computer Sci.* Springer-Verlag, 1997.

[23] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):1–21, 1996.

A Proofs for Pure Functional Language with Secrecy

A.1 Basic Facts

Proposition A.1 *Suppose* $s_1 \leq s_2$ *and* $ir_1 \sqsubseteq ir_2$. *Then* $s_1 \bullet ir_1 \leq s_2 \bullet ir_2$.

Proposition A.2 *For any secrecy property* κ *and type* s,

1. $\kappa \bullet ir \bullet ir' = \kappa \bullet ir' \bullet ir$.
2. $s \bullet ir \bullet ir' = s \bullet ir' \bullet ir$.

A.2 Substitution Lemma

Lemma A.3 *If* $\Gamma \vdash v : s'$ *and* $\Gamma, x : s' \vdash e : s$, *then* $\Gamma \vdash e[v/x] : s$.

Proof: By induction on the proof of $\Gamma, x : s' \vdash e : s$. We consider a few of the most representative cases and leave the others to the reader.

1. $\Gamma, x : s' \vdash y : s$ where $y \neq x$. Obvious.
2. $\Gamma, x : s' \vdash x : s$ where $s = s'$. Since $e[v/x] = v$, we are done.
3. $\Gamma, x : s' \vdash (\lambda y : s_1.\, e')_\kappa : (s_1 \to s_2, \kappa)$, where $\Gamma, x : s', y : s_1 \vdash e' : s_2$ and $y \neq x$. By induction, $\Gamma, y : s_1 \vdash e'[v/x] : s_2$. Thus, by rule [*Lam*],

$$\Gamma \vdash (\lambda y : s_1.\, e')_\kappa[v/x] : s$$

as desired.

This completes the induction and hence the proof. ■

A.3 Subject Reduction Theorem

Lemma A.4 *Suppose* $\emptyset \vdash e_a : s_a$, *and* $e_a \to e_b$. *Then* $\emptyset \vdash e_b : s_a$.

Proof: By cases depending on the reduction rule used. We give a few representative cases.

1. $((\lambda x : s_1.\, e)_{r,ir}\ v)_{r'} \to (\texttt{protect}_{ir'}\ e[v/x])$. By assumption, $\emptyset \vdash e_a : s_a$. Since e_a is $((\lambda x : s_1.\, e)_{r,ir}\ v)_{r'}$, this derivation must end in a (possibly empty) series of [*Sub*] applications that are immediately preceded by an application of [*App*]. Hence there exists some $s'_a \leq s_a$ and a derivation $\emptyset \vdash e_a : s'_a$ whose last rule application is [*App*]. By inspection of [*App*], there exist derivations for

$$\begin{array}{rl} & \emptyset \vdash (\lambda x : s_1.\, e)_{r,ir} : (s'_1 \to s'_2, r'', ir'') \\ & \emptyset \vdash v : s'_1 \\ \text{where} & r'' \sqsubseteq r' \text{ and } s'_2 \bullet ir'' = s'_a \leq s_a \end{array}$$

The derivation $\emptyset \vdash (\lambda x : s_1.\, e)_{r,ir} : (s'_1 \to s'_2, r'', ir'')$ must end in a (possibly empty) series of [*Sub*] applications that are immediately preceded by an application of [*Abs*]. Hence there exists an s_3 where $s_3 \leq (s'_1 \to s'_2, r'', ir'')$, and a derivation $\emptyset \vdash (\lambda x : s_1.\, e)_{r,ir} : s_3$ whose last rule is [*Abs*]. By the [*Abs*] rule, there is a derivation:

$$\begin{array}{rl} & x : s_1 \vdash e : s_2 \\ \text{where} & s_3 = (s_1 \to s_2, r, ir) \end{array}$$

Now, $s_3 = (s_1 \to s_2, r, ir) \leq (s'_1 \to s'_2, r'', ir'')$ implies that:

$$\begin{array}{l} s'_1 \leq s_1 \\ s_2 \leq s'_2 \\ ir \sqsubseteq ir'' \end{array}$$

Combining $s'_1 \leq s_1$ with $\emptyset \vdash v : s'_1$ implies $\emptyset \vdash v : s_1$ by the [*Sub*] rule. Hence we have $x : s_1 \vdash e : s_2$ and $\emptyset \vdash v : s_1$, and so by the Substitution Lemma,

$$\emptyset \vdash e[v/x] : s_2.$$

By the [*Protect*] rule, $\emptyset \vdash (\texttt{protect}_{ir}\ e[v/x]) : s_2 \bullet ir$. Now, $s_2 \leq s'_2$ and $ir \sqsubseteq ir''$ and so by Proposition A.1,

$$s_2 \bullet ir \leq s'_2 \bullet ir'' = s'_a \leq s_a$$

Hence, by (*Sub*), $\emptyset \vdash (\texttt{protect}_{ir}\ e[v/x]) : s_a$.

2. $(\texttt{proj}_i\ \langle v_1, v_2 \rangle_{(r,ir)})_{r'} \to (\texttt{protect}_{ir}\ v_i)$, where $r \sqsubseteq r'$. Because $\emptyset \vdash e_a : s_a$, there must exist $s'_a \leq s_a$ and a derivation $\emptyset \vdash (\texttt{proj}_i\ \langle v_1, v_2 \rangle_{(r,ir)})_{r'} : s'_a$ whose last rule application is [*Proj*]. By the [*Proj*] rule:

$$\begin{array}{rl} & \emptyset \vdash \langle v_1, v_2 \rangle_{(r,ir)} : (s'_1 \times s'_2, (r'', ir'')) \\ \text{where} & s'_i \bullet ir'' = s'_a \leq s_a \text{ and } r'' \sqsubseteq r' \end{array}$$

Since $\emptyset \vdash \langle v_1, v_2 \rangle_{(r,ir)} : (s'_1 \times s'_2, (r'', ir''))$, there is a derivation of $\emptyset \vdash \langle v_1, v_2 \rangle_{(r,ir)} : s_3$ ending in a use of the (*Pair*) rule, such that $s_3 \leq (s'_1 \times s'_2, (r'', ir''))$. From the (*Pair*) rule we have derivations:

$$\begin{array}{rl} & \emptyset \vdash v_1 : s_1 \\ & \emptyset \vdash v_2 : s_2 \\ \text{where} & s_3 = (s_1 \times s_2, (r, ir)) \end{array}$$

and so by the [*Protect*] rule, $\emptyset \vdash (\texttt{protect}_{ir}\ v_i) : s_i \bullet ir$. Since $s_3 = (s_1 \times s_2, (r, ir)) \leq (s'_1 \times s'_2, (r'', ir''))$,

$$\begin{array}{l} s_1 \leq s'_1 \\ s_2 \leq s'_2 \\ ir \sqsubseteq ir'' \end{array}$$

Hence $s_i \bullet ir \leq s'_i \bullet ir'' = s'_a \leq s_a$ by Proposition A.1, and so $\emptyset \vdash (\texttt{protect}_{ir}\ v_i) : s_a$ by (*Sub*).

3. $(\texttt{protect}_{ir}\ ())_\kappa) \to ()_{\kappa \bullet ir}$. Since $\emptyset \vdash e_a : s_a$, there exists $s'_a \leq s_a$ and a derivation $\emptyset \vdash (\texttt{protect}_{ir}\ ())_\kappa) : s'_a$ whose last rule is [*Protect*]. Hence, there is derivation

$$\begin{array}{rl} & \emptyset \vdash ()_\kappa : s \\ \text{where} & s \bullet ir = s'_a \leq s_a \end{array}$$

The derivation of $\emptyset \vdash ()_\kappa : s$ must consist of an application of the [*Unit*] rule followed by some number of applications of [*Sub*]. Hence $(\texttt{unit}, \kappa) \leq s$. Now, applying the [*Unit*] rule to $()_{\kappa \bullet ir}$ gives:

$$\emptyset \vdash ()_{\kappa \bullet ir} : (\texttt{unit}, \kappa \bullet ir)$$

Since $(\texttt{unit}, \kappa \bullet ir) = (\texttt{unit}, \kappa) \bullet ir$ and $(\texttt{unit}, \kappa) \leq s$, it follows from Proposition A.1 that $(\texttt{unit}, \kappa \bullet ir) \leq s \bullet ir = s'_a \leq s_a$. Hence, $\emptyset \vdash ()_{\kappa \bullet ir} : s_a$ by [*Sub*].

This concludes the case analysis and hence the proof. ■

Theorem 2.1 (Subject Reduction) *Suppose $\emptyset \vdash e : s$ and $e \to e'$. Then $\emptyset \vdash e' : s$.*

Proof: Note that $e = E[e_1]$, where $e_1 \to e_2$ via one of the rules in Table 1, and $e' = E[e_2]$. A simple induction on evaluation contexts, using Lemma A.4, completes the proof. ■

A.4 Progress Theorem

Theorem 2.2 (Progress) *Suppose $\emptyset \vdash e : s$ and e is not a value. Then there is a reduction $e \to e'$.*

Proof: Suppose, by way of contradiction, that there is no reduction of e. Then it must be the case that $e = E[e_0]$, $\emptyset \vdash e_0 : s_0$ for some s_0, and e_0 has one of the following forms:

1. $e_0 = (v\, v')_r$.
2. $e_0 = (\mathtt{proj}_i\, v)_r$.
3. $e_0 = (\mathtt{case}\, v\, \mathtt{of}\, \mathtt{inj}_1(x).\, e_1 \mid \mathtt{inj}_2(x).\, e_2)_r$.

We consider the first case and leave the others to the reader. Since e_0 is well-typed,

$$\begin{array}{l} s_0' \le s_0 \\ \emptyset \vdash (v\, v')_r : s_0' \\ \emptyset \vdash v : (s_1 \to s_0', (r_2, ir_2)) \\ \emptyset \vdash v : s_1 \end{array}$$

and $r_2 \sqsubseteq r$. Note that v must have the form $(\lambda x : s_1.\, e_0')_{(r_1, ir_1)}$, since it has a functional type (this can be seen by an easy induction on typing derivations).

This gives us enough room to complete the proof. By rule [*Abs*], we know

$$\begin{array}{l} (s_1 \to s_0, (r_1, ir_1)) \le (s_1' \to s_0', (r_2, ir_2)) \\ \emptyset \vdash_{ir} (\lambda x : s_1.\, e_0')_{(r_1, ir_1)} : (s_1 \to s_0, (r_1, ir_1)) \\ x : s_1 \vdash_{ir'} e_0' : s_0 \end{array}$$

It follows that $r_1 \sqsubseteq r_2 \sqsubseteq r$, and so the application reduction rule applies. This contradicts the initial assumption that there is no reduction of e, so there must be a reduction of the term. ■

A.5 Noninterference

We can assign a standard denotational semantics to the language by adopting the partial function model of [17]. Define the meaning of a type expression s, denoted $[\![s]\!]$, by

$$\begin{array}{rcl} [\![(\mathtt{unit}, (r, ir))]\!] & = & \{*\} \\ [\![(s + t, (r, ir))]\!] & = & ([\![s]\!] + [\![t]\!]) \\ [\![(s \times t, (r, ir))]\!] & = & ([\![s]\!] \times [\![t]\!]) \\ [\![(s \to t, (r, ir))]\!] & = & ([\![s]\!] \to_p [\![t]\!]) \end{array}$$

where $(D \to_p E)$ is the set of partial continuous functions from D to E. Note that this semantics ignores the security properties.

The meaning of terms is a *partial* function. If $\Gamma = x_1 : t_1, \ldots, x_n : t_n$ is a typing context then $[\![\Gamma]\!] = [\![t_1]\!] \times \ldots \times [\![t_n]\!]$. (The order is not important here, as we could rely on some fixed ordering of $x_i : t_i$ pairs.) In the case that Γ is empty, $[\![\Gamma]\!]$ is the unit object *unit*. For an environment $\eta \in |[\![\Gamma]\!]|$, write $\eta(x)$ for the projection to the component corresponding to variable x, and $\eta[x \mapsto d]$ for the environment in which the x component is extended (or overwritten) to d. The definition of the meaning function on terms, like that of types, ignores the security properties; similar definitions may be found in, say, [17]. The model is adequate for observing the final answers of programs:

Theorem A.5 (Plotkin) *For any typing judgement $\emptyset \vdash M : s$ and any environment η, $[\![\emptyset \vdash M : s]\!]\eta$ is defined iff $M \to^* v$ for some value v.*

Our proof of noninterference uses logical relations (see [12] for other uses of logical relations). Define R to be a family of relations indexed by secure types and indirect readers *ir* where

1. If $s = (t, (r, ir))$ and $ir \not\sqsubseteq ir'$, then $R^s_{ir'} = \{(d, e) \mid d, e \in [\![s]\!]\}$.
2. If $t = (\mathtt{unit}, (r, ir))$ and $ir \sqsubseteq ir'$, then $R^s_{ir'} = \{(*, *)\}$.
3. If $t = (s_1 + s_2, (r, ir))$ and $ir \sqsubseteq ir'$, then
$$R^s_{ir'} = \{(inj_i(d), inj_i(e)) \mid (d, e) \in R^{s_i}_{ir'}, i = 1, 2\}.$$
4. If $s = (s_1 \times s_2, (r, ir))$ and $ir \sqsubseteq ir'$, then
$$R^s_{ir'} = \{(\langle d_1, e_1\rangle, \langle d_2, e_2\rangle) \mid (d_i, e_i) \in R^{s_i \bullet ir}_{ir'}\}.$$
5. If $s = (s_1 \to s_2, (r, ir))$ and $ir \sqsubseteq ir'$, then
$$R^s_{ir'} = \{(f, g) \mid \text{if } (d, e) \in R^{s_1}_{ir'}, \text{ then } (f(d), g(e)) \in \overline{R^{s_2 \bullet ir}_{ir'}}\}.$$

Here, $(f(d), g(e)) \in \overline{R^s_{ir'}}$ means that if $f(d)$ and $g(e)$ are defined, then $(f(d), g(e)) \in R^s_{ir'}$. Intuitively, the ir' index specifies the secrecy group of an indirect reader of group ir'. When the secrecy group ir' is not above the group of the type itself, the indirect reader does not have permission to find out any information about the value.

Proposition A.6 *1. Each R^s_{ir} is directed complete. That is, if $\{(d_i, e_i) \mid i \in I\} \subseteq R^s_{ir}$ is a directed set, then $(\bigsqcup d_i, \bigsqcup e_i) \in R^s_{ir}$.*

2. If $s \le s'$, then $R^s_{ir} \subseteq R^{s'}_{ir}$.

Proof: By induction on types. ■

Theorem A.7 *Suppose $\Gamma \vdash e : s$ and $\eta, \eta' \in [\![\Gamma]\!]$. Suppose that for all $x : s' \in \Gamma$, $(\eta(x), \eta'(x)) \in R^{s'}_{ir'}$. Then $([\![\Gamma \vdash e : s]\!]\eta, [\![\Gamma \vdash e : s]\!]\eta') \in \overline{R^s_{ir'}}$.*

Proof: By induction on the proof of $\Gamma \vdash e : s$. ■

Suppose s is a type. Then s is **transparent at security property** κ if

1. $s = (\mathtt{unit}, \kappa')$ and $\kappa' \le \kappa$;
2. $s = (s_1 + s_2, \kappa')$, $\kappa' \le \kappa$, and s_1, s_2 are transparent at security property κ;
3. $s = (s_1 \times s_2, \kappa')$, $\kappa' \le \kappa$, and s_1, s_2 are transparent at security property κ; or
4. $s = (s_1 \to s_2, \kappa')$, $\kappa' \le \kappa$, and s_1, s_2 are transparent at security property κ.

$s = (t, \kappa)$ is **transparent** if s is transparent at κ.

Lemma A.8 *Suppose $s = (t, (r, ir))$ is a ground type, transparent at (r', ir'). If $(f_1, f_2) \in R^s_{ir'}$, then $f_1 = f_2$.*

Proof: By induction on t. The base case, when $t = \texttt{unit}$, is obvious. When $t = (s_1 + s_2, (r, ir))$, since $ir \sqsubseteq ir'$, it follows from the definition of $R^s_{ir'}$ that $f_j = (inj_i\ e_j)$ for some i and $(e_1, e_2) \in R^{s_i}_{ir'}$. By induction, $e_1 = e_2$. Thus, $f_1 = f_2$.

When $t = (s_1 \times s_2, (r, ir))$, it follows from the definition of $R^s_{ir'}$ that $f_j = \langle d_j, e_j \rangle$ and $(d_1, d_2) \in R^{s_1 \bullet ir}_{ir'}$ and $(e_1, e_2) \in R^{s_2 \bullet ir}_{ir'}$. Note that $s_1 \bullet ir = (t_1, (r_1, ir_1)) \bullet ir = (t_1, (r_1 \sqcup ir, ir_1 \sqcup ir))$ and similarly for $s_2 \bullet ir = (t_2, (r_2, ir_2)) \bullet ir$. Since $ir_1 \sqsubseteq ir$, $(ir_1 \sqcup ir) = ir \sqsubseteq ir'$, and similarly $(ir_2 \sqcup ir) \sqsubseteq ir'$. Thus, by induction, $d_1 = d_2$ and $e_1 = e_2$, which proves that $f_1 = f_2$ as desired. ■

Theorem 2.3 (Noninterference) *Suppose* $\emptyset \vdash e, e' : (t, (r, ir))$ *and* $\emptyset \vdash C[e] : (t', (r', ir'))$, t' *is a transparent ground type and* $ir \not\sqsubseteq ir'$. *Then* $C[e] \simeq C[e']$.

Proof: To simplify notation, let $\texttt{unit}$ stand for the least secure unit type $(\texttt{unit}, (\bot, \bot))$ (with lowest security) and $() : \texttt{unit}$ denote the least secure value of type $\texttt{unit}$. Consider the open term

$$y : (\texttt{unit} \to s, (\bot, \bot)) \vdash C[(y\ ())_\bot] : s'$$

It is easy to see that this is a well-formed typing judgement. Consider any $\emptyset \vdash e_i : s$ for $i = 1, 2$. Let

$$d_i = [\![\emptyset \vdash (\lambda x : \texttt{unit}.\, e_i)_{(\bot,\bot)} : (\texttt{unit} \to s, (\bot, \bot))]\!].$$

It is easy to show that $(d_1, d_2) \in R^{(\texttt{unit} \to s, (\bot,\bot))}_{ir'}$, since $ir \not\sqsubseteq ir'$. Let

$$f_i = [\![y : (\texttt{unit} \to s, (\bot, \bot)) \vdash C[(y\ ())_\bot] : s']\!][x \mapsto d_i].$$

By Theorem A.7,

$$(f_1, f_2) \in \overline{R^{s'}_{ir'}}.$$

If f_1, f_2 are defined, then by Lemma A.8, $f_1 = f_2$. When f_i is defined, it is simple to show that there is a value v_i such that $f_i = [\![\emptyset \vdash v_i : s']\!]$. Since $v_1 \simeq v_2$, we are done. ■