A Cryptographic Decentralized Label Model

Jeffrey A. Vaughan   Steve Zdancewic*

University of Pennsylvania

Abstract

Information-flow security policies are an appealing way of specifying confidentiality and integrity policies in information systems. Most previous work on language-based security has assumed that programs run in a closed, managed environment and that they use potentially unsafe constructs, such as declassification*, to interface to external communication channels, perhaps after encrypting data to preserve its confidentiality. This situation is unsatisfactory for systems that need to communicate over untrusted channels or use untrusted persistent storage, since the connection between the cryptographic mechanisms used in the untrusted environment and the abstract security labels used in the trusted language environment is ad hoc and unclear.*

This paper addresses this problem in three ways: First, it presents a simple, security-typed language with a novel mechanism called packages that provides an abstract means for creating opaque objects and associating them with security labels; well-typed programs in this language enforce noninterference. Second, it shows how to implement these packages using public-key cryptography. This implementation strategy uses a variant of Myers and Liskov's decentralized label model, which supports a rich label structure in which mutually distrusting data owners can specify independent confidentiality and integrity requirements. Third, it demonstrates that this implementation of packages is sound with respect to Dolev-Yao style attackers—such an attacker cannot determine the contents of a package without possessing the appropriate keys, as determined by the security label on the package.

1 Introduction

Information-flow security policies are an appealing way of specifying confidentiality and integrity policies in information systems. Unlike traditional reference monitors and cryptography, which regulate *access* to data, mechanisms that enforce information-flow policies regulate how the data (and information derived from the data) is allowed to *propagate* throughout the system. Such end-to-end security properties are important for applications that require high degrees of confidentiality (such as those found in SELinux [28]) and integrity (such as those found in critical embedded systems [9]).

Language-based mechanisms, which rely on static program analysis, are one approach to determining whether a given piece of software obeys an information-flow policy. The key idea, stemming from the work by Denning [13, 14] in the 1970's, is to annotate program values with *labels* drawn from a lattice of security levels and then have the compiler verify that the program follows the standard "no read up/no write down" noninterference policy [18, 8]. Following the work of Volpano, Smith, and Irvine [33], these program analyses are usually expressed as a form of type-checking.

The literature in this area has explored a wide variety of label models, programming language features, mechanisms for dealing with declassification and other kinds of downgrading, and appropriate definitions of security (see the survey by Sabelfeld and Myers [27]). FlowCaml [25, 29] and Jif [11] are two full-fledged programming languages that support information-flow security policies. Jif, for example, has been used to implement some simple distributed games [5, 35] and a secure e-mail system [20].

Despite these promising results, one important open question in the design of languages for information-flow security is how to integrate them with other mechanisms such as cryptography and traditional access controls. Understanding the relationship between cryptography and information-flow is particularly important in the case of "open" systems in which the data to be protected must leave the managed environment provided by the language runtime. For example, if the system needs to send protected data over an untrusted network or write it to persistent storage, encryption and digital signatures are the appropriate means of providing confidentiality and integrity.

*This research was sponsored in part by NSF Grants CNS-0346939, CNS-0524059 and CCF-0524035. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

Although cryptography is an extremely valuable tool for security engineering, there has been surprisingly little work on developing a theory of how it and information-flow mechanisms can be brought together coherently. The work in this space includes the KDLM [12], crypto-masked flows [4], sealing calculi [31], cryptographic types [17], and computational security analyses of information-flow with encryption [21, 30]. In this paper, we explore a novel design for incorporating cryptographic operations with language-based information-flow security.

We have three main goals for the programming language presented here. First, we want the programming model to have abstractions suitable for cryptographically enforcing information-flow policies specified via security labels. Second, the design of the new language primitives should free the programmer from the burden of having to manually manage keys and their correspondence to information-flow policy labels. And, third, we should prove that, under a reasonable model of cryptography, programs written in the resulting language satisfy the standard *noninterference* properties expected in this context.

In this paper, we realize the goals above by making the following contributions:

- We develop a language, SImp, with primitives for enforcing information-flow security policies, including a restricted form of cryptographic packaging. The novel language constructs are reminiscent of the pack/unpack operations found in languages with existential or dynamic datatypes. Operationally, the use of these packaging constructs requires run-time checks that ensure the security of program [17].

- We show that packages have a natural implementation in terms of public-key cryptography by defining a translation from language values to cryptographic messages; this translation depends on the structure of the labels used to define security policies. A variant of the decentralized label model [22] provides a pleasant setting for the translation.

- We prove a noninterference result for SImp, including the downgrading implicit in its cryptographic packages. We also demonstrate the soundness of the cryptographic interpretation of packages by showing that a Dolev-Yao attacker [16] cannot determine the contents of a package without possessing the appropriate keys (as determined by the translation of the security label on the package).

The rest of this paper is structured as follows. Section 2 introduces our information flow language and proves noninterference. Section 3 gives a Dolev-Yao system for reasoning about cryptography and a translation from language values to cryptographic messages. Sections 4 and 5 contain discussion and related work, respectively.

2 The SImp Language

2.1 Background and Example

As with other language-based approaches to information-flow security, the locations in our programming language are annotated with security labels. This paper uses a decentralized label model (DLM) variant where labels are lists of security policies with confidentiality and integrity components. These policies refer to principals, which are characterized by their access to private keys. A policy has form $o : \overline{r} \,!\, \overline{w}$, where $\overline{r}$ and $\overline{w}$ are sets of principals and o is a single principal. This means that policy owner o certifies that any principal in $\overline{r}$ can read from the associated location, and any principal in $\overline{w}$ can write. Sections 2.2 and 3.1 discuss the label model and private keys respectively.

Although the literature discusses "the" DLM, there are several subtly different models. When they are handled at all, integrity constraints sometimes correspond to *writers* of data; other times to *trusters*. Additionally, DLM presentations typically contain an *acts-for hierarchy*: an explicit and nominal delegation relation. Here, we do not build an explicit acts-for hierarchy as it is not germane to our setting. Instead we investigate the orthogonal issues of collusion and cooperation among sets of principals. Section 5.1 discusses the acts-for hierarchy further.

Before examining the formal description of SImp, we present the sample program shown in Figure 1. In this example we imagine a small client that can read and write data to a database shared by many users.[1] The database implements a finite map signature, with no provisions for security. To model this situation, the database is labeled with a single policy: $\{\mathit{db\_admin} : \mathit{everyone} \,!\, \mathit{everyone}\}$. That is, data entered in to the database is readable by anyone; data read from it may have been altered by anyone.

Input and output are performed by reading from and writing to designated memory locations. Lines 14 through 20 declare the locations corresponding to input, and line 23 declares a location corresponding to the terminal. Locations `action` and `position` describe the program's mode of operation—whether to read or write and where. Tagged with security label $\{p : \mathit{everyone} \,!\, p\}$, their contents are readable by everyone and have only been influenced by one principal, p. The label on `txt` is more restrictive; its contents are only readable by p. That is, `txt` contains a secret. The database is promiscuous; it produces and

[1]For convenience we have augmented SImp with syntactic sugar, method calls, strings, and a unit type.

```
 1  /* The database accepts and produces
 2   * input that is not confidential and
 3   * that anyone may have influenced. */
 4  labeldef db_io =
 5               {db_admin:everyone!everyone}
 6
 7  /* database interface */
 8  store: (int * pkg){db_io} -> (){db_io}
 9  retrieve: int{db_io} -> pkg{db_io}
10
11  /* locations representing input */
12
13  /* put = true; get = false */
14  action: bool{p:everyone!p}
15
16  /* record id */
17  position: int{p:everyone!p}
18
19  /* a confidential note */
20  txt: string{p:p!p}
21
22  /* location representing output */
23  console: string{p:p!everyone}
24
25  /* scratch location */
26  reply: pkg{db_io}
27
28  case action of
29    put =>
30      store(pos, pack txt at {p:p!p});
31      console := "text stored"
32  | get =>
33      reply := retrieve(pos);
34      case (unpack reply
35                as string{p:p!p}) of
36        inl v => console := v
37      | inr _ => console := "bad package"
```

Figure 1. An example SImp program

consumes values that, according to *db_admin*, are world-readable and have no integrity constraints.

The branches of the outer `case` command store and retrieve data from the database. In the "put" case, the client wishes to enter secret value `txt` to the database. However simply calling `store(pos, txt)` would not be secure. (Additionally, the shape of `txt` is `string` while `store` expects a `pkg`—an important detail, but only peripherally related to security.) We can deduce that this call is insecure in two ways. First, the database semantics are insecure; anyone could read `txt` if it were `store`d directly. Second, the label of `txt` specifies that p requires that only p can read, while `store` (line 8) requires arguments readable by anyone. Here a simple syntactic check of security labels identifies a semantic error; this is the point of static information-flow analysis. The actual invocation of `store` on line 30 satisfies the label checking and avoids the semantic error. It does not leak information because `pack` builds a cryptographic message which encrypts (and signs) `txt`. The typing rules reflect this, allowing the result of a `pack` to be treated as world-readable data.

In the case of a "get", `unpack`ing `reply`—the publicly readable result of `retrieve`—yields either a confidential and trusted string, or an error. As we will see, `unpack`ing requires static and dynamic checks that work in concert to prevent undesired information flows.

2.2 Security Lattice Properties

As we saw above, variables in SImp programs are annotated with security labels. The language definition is parameterized by the algebraic structure of labels and several basic axioms. This section describes the generic label properties and defines a variant of Myers and Liskov's decentralized label model (DLM) [22], a concrete instantiation of the structure. In Section 3.2 we examine how to compile SImp values into cryptographic messages; that discussion will assume labels are defined by our DLM.

Labels, denoted ℓ, are elements of a non-trivial, bounded lattice with order relation $\leq$ and join operation $\sqcup$. Upper bound $\top$ is the most restrictive label, and $\bot$ is the least restrictive. Labels have confidentiality and integrity components. A pair of functions, C and I, allow us to consider separately parts of a label; $C(\ell)$ returns a label with ℓ's confidentiality policy and the least restrictive integrity policy. Function I is the integrity analog. Both functions are idempotent. Formally,

$$\ell = C(\ell) \sqcup I(\ell) \qquad C(I(\ell)) = \bot \qquad I(C(\ell)) = \bot$$

$$C(C(\ell)) = C(\ell) \qquad\qquad I(I(\ell)) = I(\ell).$$

Additionally, we assume C and I are monotone.

$$\ell \leq \ell' \iff C(\ell) \leq C(\ell') \wedge I(\ell) \leq I(\ell')$$

The purpose of labels is to classify who can read, and who could have written, data. We assume that there is a fixed set of principals, $\mathcal{P}$, ranged over by p. We also require two monotone predicates that indicate whether a set of principals, $\overline{p} \subseteq \mathcal{P}$, can read (resp. write) according to a label's confidentiality (integrity) component. Formally, if $C(\ell) \leq C(\ell')$ then $\overline{p}$ *reads* ℓ' implies $\overline{p}$ *reads* ℓ. Integrity is the opposite: if $I(\ell) \leq I(\ell')$ then $\overline{p}$ *writes* ℓ implies $\overline{p}$ *writes* ℓ'. We call a label set and operators over that set a security lattice when the above proprieties hold.

We instantiate the above with a decentralized label model that omits the acts-for hierarchy [22] and assumes that principals can collude to pool their authority. That is, we intend for $\overline{p}$ *reads* ℓ (resp. $\overline{p}$ *writes* ℓ) to hold when the members of $\overline{p}$ can *cooperate* to read (write) at ℓ. Section 5.1 compares our presentation of a DLM with several others, including Myers and Liskov's original description.

In a DLM, principals typically represent users of a system. We call the set of all principals $\mathcal{P}$, and assume it is finite. We also assume the existence of a canonical total ordering on $\mathcal{P}$; this is not the acts-for hierarchy, but a helpful condition used for defining functions over *labels*.

Informally, a label consists of several policies in which principals called owners make access control statements. Each policy has form $o : \overline{r}\,!\,\overline{w}$, and consists of an owner o, a set of readers $\overline{r}$, and a set of writers $\overline{w}$. When attached to a piece of data, such a policy means that owner o certifies that the data can be read only with the authority of o or of some principal in $\overline{r}$. Similarly, p can only write such data when p is o or in $\overline{w}$.

Formally, a DLM label is function of type $\mathcal{P} \to 2^{\mathcal{P}} \times 2^{\mathcal{P}}$. Notation $\ell = \{o_1 : \overline{r}_1\,!\,\overline{w}_1; o_2 : \overline{r}_2\,!\,\overline{w}_2\}$ abbreviates

$$\ell(o) = \begin{cases} (\overline{r}_1, \overline{w}_1) & o = o_1 \\ (\overline{r}_2, \overline{w}_2) & o = o_2 \\ (\mathcal{P}, \emptyset) & \text{otherwise.} \end{cases}$$

The projections $\ell(o_1).C$ and $\ell(o_2).I$ give $\overline{r}_1$ and $\overline{w}_2$. Because principals are totally ordered, functions (predicates) may be defined by recursion (induction) as if labels were lists of policies.

The inequality $\ell_1 \leq \ell_2$ holds when

$$\forall o.\ \ell_2(o).C \subseteq \ell_1(o).C \ \wedge\ \ell_1(o).I \subseteq \ell_2(o).I.$$

The confidentiality of the policy defined by $C(o : \overline{r}\,!\,\overline{w}) = o : \overline{r}\,!\,\emptyset$; integrity is defined by $I(o : \overline{r}\,!\,\overline{w}) = o : \mathcal{P}\,!\,\overline{w}$. These definitions generalize to labels in the natural way. If $\ell = \ell_1 \sqcup \ell_2$ then

$$\ell(o) \quad = \quad (\ell_1(o).C \cap \ell_2(o).C,\ \ell_1(o).I \cup \ell_2(o).I).$$

Predicates for reading and writing hold when principals can cooperate to read or write labeled data. Predicate $\overline{p}$ *reads* ℓ is defined to be true iff $\forall o.\ \exists p \in \overline{p}.\ p \in \{o\} \cup \ell(o).I$, and $\overline{p}$ *writes* ℓ when $\forall o.\ \exists p \in \overline{p}.\ p \in \{o\} \cup \ell(o).C$. Intuitively, $\overline{p}$ can read (write) when every owner permits at least one member of $\overline{p}$ to read (write).

As a notational convenience, we will write $\{p_1, p_2 : \overline{r}\,!\,\overline{w}\}$ for $\{p_1 : \overline{r}\,!\,\overline{w}; p_2 : \overline{r}\,!\,\overline{w}\}$. It's clear that the most restrictive label, $\{\top\}$, is $\{\mathcal{P} : \emptyset\,!\,\mathcal{P}\}$, and the least restrictive label, $\{\bot\}$, is $\{\mathcal{P} : \mathcal{P}\,!\,\emptyset\}$.

Lemma 1. *The DLM is an instance of a security lattice.*

2.3 SImp Syntax

SImp is based on Winskel's IMP language [34] and the simple security language by Volpano, Smith, and Irvine [33]. SImp is stratified into pure expressions and imperative commands. We examine the language starting with syntax, then work from dynamic to static semantics.

The following grammar gives the syntax of SImp:

$$\begin{array}{llll}
\text{Types} & \tau & ::= & \texttt{int} \mid \tau_1 + \tau_2 \mid \texttt{pkg} \\
\text{Integers} & i & ::= & \ldots -1 \mid 0 \mid 1 \ldots \\
\text{Values} & v & ::= & i \mid \texttt{inl}\ v \mid \texttt{inr}\ v \mid \langle v \rangle_\ell \\
\text{Expressions} & e & ::= & i \mid a \mid x \mid \texttt{inl}\ e \mid \texttt{inr}\ e \mid e_1 + e_2 \\
 & & \mid & \langle v \rangle_\ell \mid \texttt{pack}\ e\ \texttt{at}\ \ell \\
 & & \mid & \texttt{unpack}\ e\ \texttt{as}\ \tau\{\ell\} \mid \ldots \\
\text{Commands} & c & ::= & \texttt{skip} \mid x := e \mid c_1; c_2 \\
 & & \mid & \texttt{while}\ e\ \texttt{do}\ c \\
 & & \mid & \texttt{case}\ e\ \texttt{of}\ a_1 \Rightarrow c_1 \mid a_2 \Rightarrow c_2
\end{array}$$

Expressions may of course be augmented with additional operations on `int`s as necessary. Pairs of types and labels, written $\tau\{\ell\}$, describe the shape of an expression and its security policy. Such pairs are called labeled types. We distinguish variables from locations. Variables, ranged over by a, are bound in case commands and replaced by substitution. Locations, ranged over by x, are never substituted away and are used to read from and write to memory.

The new constructs for abstract encryption include packages $\langle v \rangle_\ell$, `pack`, and `unpack`. The package $\langle v \rangle_\ell$ is intended to have several properties:

1. v must only be read by programs with sufficient authority to read ℓ.

2. v must be kept confidential in accordance with $C(\ell)$.

3. v must only be written to by programs with authority to write ℓ.

4. v must have only been influence by data with integrity greater than $I(\ell)$.

The third property restricts package creation in SImp; it would be a more powerful statement if SImp supported first-class pointers and structures.

Expression `pack` v `at` ℓ constructs package $\langle v \rangle_\ell$. Expression `unpack` v `as` $\tau\{\ell\}$ attempts to interpret v as a package $\langle v_0 \rangle_{\ell_0}$ where v_0 has shape τ and where $\ell_0 \leq \ell$. Logically, `pack` and `unpack` serve as introduction and elimination rules for `pkg`. Expression forms `pack` e `at` ℓ and $\langle v \rangle_\ell$ are not redundant—`pack` is an expression that may fail at runtime; $\langle v \rangle_\ell$ is the result of a successful `pack`.

No primitive type describes booleans or errors, but we encode them with the following abbreviations:

$$\texttt{bool} = \texttt{int} + \texttt{int}$$

$$\texttt{true}_i = \texttt{inl}\ i \qquad\qquad \texttt{false}_i = \texttt{inr}\ i$$

$$\texttt{error} = \texttt{int} \qquad \texttt{insufficientAuth} = 0$$

$$\texttt{illegalFlow} = 1 \qquad \texttt{typeMismatch} = 2$$

Typically the index of `true` or `false` is unimportant and will be omitted. Additionally, `if` e `then` c_1 `else` c_2 is shorthand for `case` e `of` $a_1 \Rightarrow c_1 \mid a_2 \Rightarrow c_2$ where a_1 and a_2 do not appear in c_1 or c_2.

2.4 Dynamic Semantics

SImp programs are run with the authority of some set of principals. Intuitively, a program run with Alice's authority can sign and decrypt with her private key. Authority is represented by a set of principals and appears in the $\overline{p}$ component of the command and expression evaluation rules (Figures 2 and 3).

Expressions do not have side effects but can read memory. Thus expressions must be evaluated in contexts containing a memory, M. Memories are finite maps from locations to values. Most expressions, but not `pack` and `unpack`, are standard.

As show by rules EE-PACK-OK and EE-PACK-FAIL, expression `pack` v `at` ℓ may evaluate in two ways. If dynamic check $\overline{p}$ *writes* ℓ succeeds, then the program is running with sufficient authority to write at ℓ. In this case, `pack` evaluates to `inl` $\langle v\rangle_\ell$. However, if the program cannot write at ℓ, an error results instead. While the dynamic behavior of `pack` is influenced by the *writes* relation, it is not a covert channel. The authority set $\overline{p}$, label ℓ in the text of `pack`, and the definition of *writes* do not vary at run time. Therefore an attacker cannot gain information by observing whether a `pack` succeeds.

Unpacks can fail in more ways than packs; this is reflected in the three premises of EE-UNPACK-OK. Analogously to packing, `unpack` $\langle v_0\rangle_{\ell_0}$ `as` $\tau\{\ell\}$ requires that $\overline{p}$ *reads* ℓ. However, the contents and label of $\langle v_0\rangle_{\ell_0}$ are statically unknown, and we must make two additional runtime checks. First, checking $\ell_0 \leq \ell$ ensures that the static information flow properties of SImp continue to protect v_0 after unpacking. Second, checking $\vdash v_0 : \tau$ (which typechecks v_0 to ensure it has type τ) is required to avoid dynamic type errors. This check must be delayed until runtime; checking sooner is incompatible with the cryptographic semantics given in Section 3.2.

The only unusual command is `case`. Tagged unions, such as `inl` 0 are consumed by `case`, which branches on the tag (i.e. `inl`) and substitutes the value (i.e. 0) for a bound variable in the taken branch. Evaluation rules EC-CASEL and EC-CASER define this behavior.

2.5 Static Semantics

SImp's type system performs two roles. First, it provides type safety; this means that the behavior of well typed programs is always defined. Second, the type system prevents high-to-low information flows except where permitted by

$$\boxed{\overline{p}; M \vdash e \to e'}$$

$$\textsc{ee-Loc}\quad \frac{M(x) = v}{\overline{p}; M \vdash x \to v} \qquad \textsc{ee-Inl}\quad \frac{\overline{p}; M \vdash e \to e'}{\overline{p}; M \vdash \texttt{inl}\ e \to \texttt{inl}\ e'}$$

$$\frac{\overline{p}; M \vdash e \to e'}{\overline{p}; M \vdash \texttt{inr}\ e \to \texttt{inr}\ e'}\ \textsc{ee-Inr}$$

$$\frac{\overline{p}; M \vdash e_1 \to e_1'}{\overline{p}; M \vdash e_1 + e_2 \to e_1' + e_2}\ \textsc{ee-Plus-Struct1}$$

$$\frac{\overline{p}; M \vdash e \to e'}{\overline{p}; M \vdash v + e \to v + e'}\ \textsc{ee-Plus-Struct2}$$

$$\frac{}{\overline{p}; M \vdash i_1 + i_2 \to i_3}\ \textsc{ee-Plus} \quad \text{where } [\![i_3 = i_1 + i_2]\!]$$

$$\frac{\overline{p}; M \vdash e \to e'}{\overline{p}; M \vdash \texttt{pack}\ e\ \texttt{at}\ \ell \to \texttt{pack}\ e'\ \texttt{at}\ \ell}\ \textsc{ee-Pack-Struct}$$

$$\frac{\overline{p}\ \mathit{writes}\ \ell}{\overline{p}; M \vdash \texttt{pack}\ v\ \texttt{at}\ \ell \to \texttt{inl}\ \langle v\rangle_\ell}\ \textsc{ee-Pack-OK}$$

$$\textsc{ee-Pack-Fail}\quad \frac{\neg(\overline{p}\ \mathit{writes}\ \ell)}{\overline{p}; M \vdash \texttt{pack}\ v\ \texttt{at}\ \ell \to \texttt{inr insufficientAuth}}$$

$$\textsc{ee-Unpack-Struct}\quad \frac{\overline{p}; M \vdash e \to e'}{\overline{p}; M \vdash \texttt{unpack}\ e\ \texttt{as}\ \tau\{\ell\} \to \texttt{unpack}\ e'\ \texttt{as}\ \tau\{\ell\}}$$

$$\textsc{ee-Unpack-Fail1}\quad \frac{\neg(\overline{p}\ \mathit{reads}\ \ell)}{\begin{array}{l}\overline{p}; M \vdash \texttt{unpack}\ \langle v_0\rangle_{\ell_0}\ \texttt{as}\ \tau\{\ell\} \to \\ \qquad\qquad \texttt{inr insufficientAuth}\end{array}}$$

$$\textsc{ee-Unpack-Fail2}\quad \frac{\overline{p}\ \mathit{reads}\ \ell \qquad \ell_0 \not\leq \ell}{\overline{p}; M \vdash \texttt{unpack}\ \langle v_0\rangle_{\ell_0}\ \texttt{as}\ \tau\{\ell\} \to \texttt{inr illegalFlow}}$$

$$\textsc{ee-Unpack-Fail3}\quad \frac{\overline{p}\ \mathit{reads}\ \ell \qquad \ell_0 \leq \ell \qquad \not\vdash v : \tau}{\overline{p}; M \vdash \texttt{unpack}\ \langle v_0\rangle_{\ell_0}\ \texttt{as}\ \tau\{\ell\} \to \texttt{inr typeMismatch}}$$

$$\textsc{ee-Unpack-OK}\quad \frac{\overline{p}\ \mathit{reads}\ \ell \qquad \ell_0 \leq \ell \qquad \vdash v_0 : \tau}{\overline{p}; M \vdash \texttt{unpack}\ \langle v_0\rangle_{\ell_0}\ \texttt{as}\ \tau\{\ell\} \to \texttt{inl}\ v_0}$$

Figure 2. Expression Evaluation Relation

$$\boxed{\overline{p} \vdash \langle M, c\rangle \to \langle M', c'\rangle}$$

$$\frac{\overline{p}; M \vdash e \to^* v}{\overline{p} \vdash \langle M, x := e\rangle \to \langle M[x \mapsto v], \texttt{skip}\rangle}\ \text{EC-ASSIGN}$$

$$\frac{}{\overline{p} \vdash \langle M, \texttt{skip}; c\rangle \to \langle M, c\rangle}\ \text{EC-SEQ-SKIP}$$

$$\frac{\overline{p} \vdash \langle M, c_1\rangle \to \langle M', c_1'\rangle}{\overline{p} \vdash \langle M, c_1; c_2\rangle \to \langle M', c_1'; c_2\rangle}\ \text{EC-SEQ-STRUCT}$$

EC-WHILE-FALSE
$$\frac{\overline{p}; M \vdash e \to^* \texttt{false}_i}{\overline{p} \vdash \langle M, \texttt{while}\ e\ \texttt{do}\ c\rangle \to \langle M, \texttt{skip}\rangle}$$

EC-WHILE-TRUE
$$\frac{\overline{p}; M \vdash e \to^* \texttt{true}_i}{\overline{p} \vdash \langle M, \texttt{while}\ e\ \texttt{do}\ c\rangle \to \langle M, c; \texttt{while}\ e\ \texttt{do}\ c\rangle}$$

EC-CASEL
$$\frac{\overline{p}; M \vdash e \to^* \texttt{inl}\ v}{\overline{p} \vdash \langle M, \texttt{case}\ e\ \texttt{of}\ a_1 \Rightarrow c_1 \mid a_2 \Rightarrow c_2\rangle \to \langle M, [v/a_1]c_1\rangle}$$

EC-CASER
$$\frac{\overline{p}; M \vdash e \to^* \texttt{inr}\ v}{\overline{p} \vdash \langle M, \texttt{case}\ e\ \texttt{of}\ a_1 \Rightarrow c_1 \mid a_2 \Rightarrow c_2\rangle \to \langle M, [v/a_2]c_2\rangle}$$

Figure 3. Command Evaluation Relation

pack and unpack operations. Expressions and commands are both typed using contexts. Location contexts, written Θ, map locations to labeled types. Variable contexts, Γ, map variables to labeled types.

The expression typing judgment $\Theta; \Gamma \vdash e : \tau\{\ell\}$ means that with contexts Θ and Γ expression e has shape τ and can be given label ℓ. Expressions relate to at most one shape, but may be assigned many different labels. This is made explicit by TE-SUB, which allows an expression's label to be raised arbitrarily. In contrast, no rule lowers labels. Rule TE-VAR looks up locations in Θ and assigns corresponding labels to locations; this standard rule prevents read up.

The typing of pack and unpack is novel. Expression $\texttt{pack}\ e\ \texttt{at}\ \ell$ can be assigned a label only when e has label ℓ. Rule TE-PACK gives $\texttt{pack}\ e\ \texttt{at}\ \ell$ label ℓ, subject to the constraint $I(\ell) = I(\ell_e)$ where ℓ_e classifies e. This is because a successful pack will yield a package that can only be deciphered by code with authority sufficient to read ℓ. Therefore it is safe to assign the resulting package an arbitrary confidentiality policy. Because packing does not attempt endorse e, integrity is preserved unchanged.

$$\boxed{\Theta; \Gamma \vdash e : \tau\{\ell\}}$$

$$\frac{}{\Theta; \Gamma \vdash i : \texttt{int}\{\ell\}}\ \text{TE-INT}$$

$$\frac{}{\Theta; \Gamma \vdash \langle v\rangle_{\ell'} : \texttt{pkg}\{\ell\}}\ \text{TE-PACKAGE}$$

$$\frac{\Theta; \Gamma \vdash e_1 : \texttt{int}\{\ell\} \qquad \Theta; \Gamma \vdash e_2 : \texttt{int}\{\ell\}}{\Theta; \Gamma \vdash e_1 + e_2 : \texttt{int}\{\ell\}}\ \text{TE-PLUS}$$

$$\frac{\Theta; \Gamma \vdash e : \tau_1\{\ell\}}{\Theta; \Gamma \vdash \texttt{inl}\ e : (\tau_1 + \tau_2)\{\ell\}}\ \text{TE-SUML}$$

$$\frac{\Theta; \Gamma \vdash e : \tau_2\{\ell\}}{\Theta; \Gamma \vdash \texttt{inr}\ e : (\tau_1 + \tau_2)\{\ell\}}\ \text{TE-SUMR}$$

$$\frac{\Theta; \Gamma \vdash e : \tau\{\ell_e\} \qquad I(\ell_e) = I(\ell)}{\Theta; \Gamma \vdash \texttt{pack}\ e\ \texttt{at}\ \ell : (\texttt{pkg} + \texttt{error})\{\ell\}}\ \text{TE-PACK}$$

$$\frac{\Theta; \Gamma \vdash e : \texttt{pkg}\{\ell_e\} \qquad C(\ell_e) = C(\ell)}{\Theta; \Gamma \vdash \texttt{unpack}\ e\ \texttt{as}\ \tau\{\ell\} : (\tau + \texttt{error})\{\ell\}}\ \text{TE-UNPACK}$$

$$\frac{\Gamma(a) = \tau\{\ell\}}{\Theta; \Gamma \vdash a : \tau\{\ell\}}\ \text{TE-VAR} \qquad \frac{\Theta(x) = \tau\{\ell\}}{\Theta; \Gamma \vdash x : \tau\{\ell\}}\ \text{TE-LOC}$$

$$\frac{\Theta; \Gamma \vdash e : \tau\{\ell'\} \qquad \ell' \leq \ell}{\Theta; \Gamma \vdash e : \tau\{\ell\}}\ \text{TE-SUB}$$

Figure 4. Expression Typing

Typing unpack is dual to pack. The expression $\texttt{unpack}\ e\ \texttt{as}\ \tau\{\ell\}$ is classified by ℓ when e is labeled by ℓ_e and $C(\ell_e) = C(\ell)$. That is, unpack maintains e's confidentiality but evaluates to a (potentially) lower—more trusted—integrity level. During execution e evaluates to a package of form $\langle v_0\rangle_{\ell_0}$ and conditions $\ell_0 \leq \ell$ and $\vdash v_0 : \tau$ are checked. These conditions ensure that labeled type $\tau\{\ell\}$ can classify v_0 without introducing illegal flows or stuck evaluation states. Thus the unpack can safely be given labeled type $\tau\{\ell\}$.

Command typing is basically standard. Intuitively, if judgment $pc; \Theta; \Gamma \vdash c$ holds then command c does not leak information. The pc component indicates the highest label assigned to locations or variables which may have influenced control flow at command c. First consider rule TC-ASSIGN; it only types $x := e$ when the label of x is greater than the pc joined with the label of e. This prevents write down. Now consider $\texttt{while}\ x\ \texttt{do}\ c$. Rule TC-WHILE accepts this command only when c_0 can be checked with

$$\boxed{pc; \Theta; \Gamma \vdash c}$$

$$\frac{}{pc; \Theta; \Gamma \vdash \texttt{skip}}\ \textsc{tc-Skip}$$

$$\frac{\Theta; \Gamma \vdash e : \tau\{\ell_e\} \quad \Theta(x) = \tau\{\ell\} \qquad \ell_e \sqcup pc \leq \ell}{pc; \Theta; \Gamma \vdash x := e}\ \textsc{tc-Assign}$$

$$\frac{pc; \Theta; \Gamma \vdash c_1 \qquad pc; \Theta; \Gamma \vdash c_2}{pc; \Theta; \Gamma \vdash c_1; c_2}\ \textsc{tc-Seq}$$

$$\frac{\Theta; \Gamma \vdash e : \texttt{bool}\{\ell\} \qquad pc \sqcup \ell; \Theta; \Gamma \vdash c}{pc; \Theta; \Gamma \vdash \texttt{while}\ e\ \texttt{do}\ c}\ \textsc{tc-While}$$

$$\frac{\Theta; \Gamma \vdash e : (\tau_1 + \tau_2)\{\ell\} \quad pc \sqcup \ell; \Theta; \Gamma[a_1 \mapsto \tau_1\{\ell\}] \vdash c_1 \quad pc \sqcup \ell; \Theta; \Gamma[a_2 \mapsto \tau_2\{\ell\}] \vdash c_2}{pc; \Theta; \Gamma \vdash \texttt{case}\ e\ \texttt{of}\ a_1 \Rightarrow c_1 \mid a_2 \Rightarrow c_2}\ \textsc{tc-Case}$$

$$\frac{pc'; \Theta; \Gamma \vdash c \qquad pc \leq pc'}{pc; \Theta; \Gamma \vdash c}\ \textsc{tc-Subs}$$

Figure 5. Command Typing

program counter greater than the label of x. Treating the pc in this manner is necessary to protect against implicit flows. We also type `case` commands in this manner.

The following program, which is rejected by the type system, has a dynamic information leak.

```
1  if h then
2    x := pack 0 at {⊥}
3  else
4    x := pack 0 at {⊤};
5  case (unpack x as int{⊥}) of
6    inl _ => output := 1  /* h true */
7    inr _ => output := 0  /* h false */
```

We assume `h` has label $\{\top\}$ and `output` has label $\{\bot\}$. If `h` is `true` then unpacking succeeds, and `output` is assigned 1. In contrast, if `h` is `false` then `output` is assigned 0. Thus an attacker, who is authorized to read $\{\bot\}$ but not $\{\top\}$, could determine `h` by observing `output`.

Fortunately the above program cannot be typed. Assume ℓ_x classifies `x`. Rule TC-CASE forces the pc to be $\{\top\}$ at line 2, and TC-ASSIGN requires $pc \leq \ell$. Thus $\{\top\} \leq \ell_x$ and $\ell_x = \{\top\}$. On line 5 expression `unpack` x `as int`$\{\bot\}$ can be assigned—by TE-UNPACK—label ℓ_0 where $C(\ell_0) = C(\ell_x) = C(\{\top\})$. Rule TC-CASE requires that, on line 6, $\ell_0 \leq pc$. By TC-ASSIGN, $pc \leq \{\bot\}$; so $\ell_0 = \{\bot\}$. As is no way to satisfy both $\ell_0 = \bot$ and $C(\ell_0) = C(\{\top\})$, the program is rejected.

$$\boxed{v_1 \cong_\ell v_2}$$

$$\frac{}{i \cong_\ell i}\ \textsc{ve-Int} \qquad \frac{v_1 \cong_\ell v_2}{\texttt{inl}\ v_1 \cong_\ell \texttt{inl}\ v_2}\ \textsc{ve-Inl}$$

$$\frac{v_1 \cong_\ell v_2}{\texttt{inr}\ v_1 \cong_\ell \texttt{inr}\ v_2}\ \textsc{ve-Inr}$$

$$\frac{v_1 \cong_\ell v_2}{\langle v_1 \rangle_{\ell_1} \cong_\ell \langle v_2 \rangle_{\ell_1}}\ \textsc{ve-Pack-In}$$

$$\frac{\ell_1 \not\leq \ell}{\langle v_1 \rangle_{\ell_1} \cong_\ell \langle v_2 \rangle_{\ell_1}}\ \textsc{ve-Pack-Lab}$$

Figure 6. Equivalent Values

2.6 Noninterference

This section establishes the noninterference information-flow property for well-typed SImp programs. Consider the terminating execution of program c in memory M_1 with result M_1'. Now change some high security components of M_1 to obtain M_2 and, starting in M_2, rerun c to get M_2'. If the low security parts of M_1' are different from those M_2' then c has leaked information. Instead, we would like to prove that the semantics of SImp ensure M_1' and M_2' are equivalent; that is, they differ only in their high security components. This property, noninterference, will be defined and formalized in the remainder of this section.

To make noninterference precise, we define when two values are equivalent at a label. This is done inductively with the rules in Figure 6. The most interesting rule is VE-PACK-LAB. It states that an observer at label ℓ cannot distinguish packages sealed at ℓ_1 when $C(\ell_1) \not\leq C(\ell)$. We must also define when two commands or expressions are equivalent. We elide the inductive definitions of these relations. Intuitively $e \cong_\ell e'$ holds when e and e' are equivalent values, or identical productions applied to equivalent expressions. The definition of $c \cong_\ell c'$ is similar.

Conventionally, noninterference treats memories as equivalent when the contents of corresponding low security locations are identical. We generalize this and allow equivalent memories to map low locations to equivalent, but not identical, values.

Definition 1 ($\Theta \vdash M_1 \cong_\ell M_2$)**.** *In context Θ, memories M_1 and M_2 are equivalent to an observer at level ℓ, written $\Theta \vdash M_1 \cong_\ell M_2$, when $dom(M_1) = dom(M_2)$ and $\forall x \in dom(M_1).\ \exists \tau, \ell'.\ \Theta(x) = \tau\{\ell'\} \wedge (M_1(x) \cong_\ell M_2(x) \vee \ell' \not\leq \ell)$.*

To avoid reasoning about stuck execution states, we assume commands are always run in well typed memories. Eliminating stuck states (i.e. proving type soundness) both simplifies SImp's metatheory and eliminates attacks through the runtime-error covert channel.

Definition 2 ($\Theta \vdash M\ OK$)**.** *A memory, M, is well typed in location context Θ, when $\forall x \in dom(\Theta).\ \Theta(x) = \tau\{\ell\} \wedge \vdash M(x) : \tau$. This property is written $\Theta \vdash M\ OK$.*

A trace is a set of principals and a sequence of configurations, $(\overline{p}, \langle M_1, c_1\rangle, \langle M_2, c_2\rangle, \ldots, \langle M_n, c_n\rangle)$, where for each $i \in \{1, n-1\}$, $\overline{p} \vdash \langle M_i, c_i\rangle \rightarrow \langle M_{i+1}, c_{i+1}\rangle$.

Lemma 2 (Trace determinancy)**.** *There is at most one shortest trace of form $(\overline{p}, \langle M_1, c_1\rangle, \ldots, \langle M_n, c_n\rangle)$.*

If such a trace exists, we will write it as $\overline{p} \vdash \langle M_1, c_1\rangle \rightarrow^* \langle M_2, c_2\rangle$.

The two following non-interference theorems are the primary language-theoretic results for SImp. The first states that if ℓ-equivalent expressions, e_1 and e_2, can be typed with label ℓ, then evaluating them in ℓ-equivalent memories, M_1 and M_2, yields ℓ-equivalent results, v_1 and v_2. That is, evaluation of low-security expressions occurs independently of high security inputs.

Theorem 1 (Expression Non-interference)**.** *If*

- $\Theta \vdash M_1\ OK$*,* $\Theta \vdash M_2\ OK$ *and* $\Theta \vdash M_1 \cong_\ell M_2$
- $\Theta; \cdot \vdash e_1 : \tau\{\ell_e\}$ *and* $e_1 \cong_\ell e_2$ *where* $\ell_e \leq \ell$
- $\overline{p}; M_1 \vdash e_1 \rightarrow^* v_1$ *and* $\overline{p}; M_2 \vdash e_2 \rightarrow^* v_2$

then $v_1 \cong_\ell v_2$.

The second theorem extends this to configurations, and states that ℓ-equivalence is preserved by terminating computations. These theorems are more general than those of Smith and Volpano [33] who do not account for equivalent but unequal values. Additionally, they are substantially simpler than the the robust declassification and qualified robustness theorems needed in the case of general endorsement or declassification [10, 23].

Theorem 2 (Command Noninterference)**.** *If*

- $\Theta \vdash M_1\ OK$*,* $\Theta \vdash M_2\ OK$ *and* $\Theta \vdash M_1 \cong_\ell M_2$
- $pc; \Theta; \cdot \vdash c_1$ *and* $c_1 \cong_\ell c_2$
- $\overline{p} \vdash \langle M_1, c_1\rangle \rightarrow^* \langle M_1', \texttt{skip}\rangle$ *and* $\overline{p} \vdash \langle M_2, c_2\rangle \rightarrow^* \langle M_2', \texttt{skip}\rangle$

then $\Theta \vdash M_1' \cong_\ell M_2'$.

The language level definitions and theorems treat confidentiality and integrity uniformly. This is a reflection of a well known duality between "tainted" and "secret" values and "untainted" and "public" values. This duality also arises in the cryptographic semantics described next, where confidentiality is enforced via encryption and integrity is enforced via digital signatures.

3 Cryptographic Semantics

The above noninterference property ensures security for programs run in a trusted environment. We wish to also consider hostile environments: Can we interpret labels using cryptography to ensure information flow guarantees hold in open systems?

To examine this issue, we first define a formal syntax of messages and a Dolev-Yao deduction system for reasoning about them. Next, we show how to compile SImp values into messages and establish that compiled packages implement appropriate confidentiality policies. Last, we relate memories and commands to messages in order to define and prove a cryptographic noninterference theorem. Note that we rely on the soundness of Dolev-Yao reasoning with respect to computational bounded attackers in the style of Abadi and Rogaway [3] and of Backes and Pfitzmann [6].

We make several general assumptions. Each principal has a corresponding public/private key pair. All public keys are known to all principals, and private keys are known only to the corresponding principal. Key distribution and name binding are orthogonal (but important) problems that are not considered here.

3.1 Messages and Message Analysis

This section defines messages, cryptographic states, and an inference system for reasoning about them. Messages are the basic objects in the cryptographic semantics. Cryptographic states are collections of messages that represent knowledge, ability, and belief. Lastly the inference system describes when new messages can be synthesized from an cryptographic state.

Messages are defined by the following grammar:

Principals	p, q, r	::=	Alice | Bob | ...
Key Id	κ, W, R		*(*abstract*)*
Private Keys	K^-	::=	K^-_κ
Public Keys	K^+	::=	K^+_κ
Strings	*str*	::=	"a"|"b" | ...
Messages	m, n	::=	*str* | K | p | (m, m')
		|	$enc(K, m)$
		|	$sign(K, m)$

The metavariable K ranges over both public and private keys. Message $enc(K, m)$ means m encrypted by K, and $sign(K, m)$ means m signed by K. Messages are paired with (m, m'). Public and private keys that share a κ are inverses. Lists, $[m_1, m_2, \ldots, m_n]$, are defined by nested pairs, $(m_1, (m_2, \ldots (m_n, \texttt{""}) \ldots))$. We will write $[m_1 \ldots m_k] +\!\!+ [m_{k+1} \ldots m_j]$ for $[m_1 \ldots m_j]$. Lastly, we will use $\texttt{"i"}$ and $\texttt{"o : }\overline{\texttt{r}}\texttt{ ! }\overline{\texttt{w}}\texttt{"}$ to denote the strings encoding integer value i and policy $o : \overline{r} \,!\, \overline{w}$ respectively.

We introduce a modal natural deduction style system for reasoning about a principal's knowledge. Cryptographic states, written σ, serve as contexts for the deduction system and track a principal's knowledge, abilities, and beliefs. It might be that a principal $knows$, $actswith$, or $believes$ a message. The judgment $\sigma \vdash_d m$ has the intended interpretation that message m can be derived from the contents of σ. The judgment $\sigma \vdash_u m$ has the intended interpretation that σ can use message m. And the judgment $\sigma \vdash_b m$ has the intended interpretation that m is considered trusted by σ. Generally, $\sigma \vdash_u m$ or $\sigma \vdash_b m$ are only interesting when m is a key. If σ uses a principal's private key, it has the Principal's authority; if σ believes the principal's public key, it trusts the principal. Figure 7 gives the inference rules.

We distinguish $knows$ from $actswith$ because $\sigma \vdash_d m$ implies $\sigma \vdash_u m$, but the converse is not true. Thus $actswith$ provides a convenient way to model private keys which are used, but never disclosed. Earlier, we said the relation $\overline{p} \vdash \langle \_, \_ \rangle \rightarrow \langle \_, \_ \rangle$ represents evaluation with authority $\overline{p}$; cryptographically speaking, execution requires a σ where $\sigma \vdash_u K_p^-$ for all $p \in \overline{p}$. Additionally, the belief modality allows us to model low integrity data. Cryptographically speaking, all messages signed with "untrusted" keys—with any K where $\sigma \not\vdash_b K$—will be considered uninformative, and therefore equivalent. We make these ideas precise in Section 3.3.

3.2 Compiling Policies

Generally we will encode policies by generating a series of fresh public key pairs. Plain text is encrypted and signed using the fresh keys, and a message is created by appending label information to the ciphertext. We aim to do this in such a way that principal set $\overline{p}$ can read the cryptographic interpretation of $\langle v \rangle_\ell$ text iff $\overline{p}$ *reads* ℓ. And, writing v requires the keys of $\overline{q}$ where $\overline{q}$ *writes* ℓ.

DLM labels are intended for situations of mutual distrust. When encoding a policy cryptographically, we have a high level choice to make: *Given principals Alice and Bob, should it possible for Bob to specify Alice's policy?* Concretely, whose authority should be required to create a value labeled by $\{\text{Alice} : \text{Bob} \,!\, \emptyset\}$?

Two apparently reasonable answers to this question are,

1. Policies may be created with no authority.

2. Policy creation requires the owner's authority.

The first approach is flawed. If no particular authority is required to labeled values, any user can attach arbitrary assertions to a package. Thus Eve can spoof labels and cause violations of confidentiality and integrity policies. This both enables easy attacks and muddies the theory. Therefore we follow the second approach.

Translating $\langle v \rangle_\ell$ into a message takes three steps. First we compile each policy in ℓ to a *seal* which can be used to ensure the confidentiality and integrity of a sealed message. Second, we compose the seals to create an *envelope* which can be read and written only in accordance with ℓ's meaning. Third, we translate v and write its translation into the envelope.

As we will see, envelopes serve as one way secure channels. Public key cryptography is essential to this. Each of an envelope's seals is associated with two private keys: one for reading and one writing. If a principal possesses all of the read keys, that principal can read from the envelope. Likewise, a principal possessing all the write keys can write to the envelope. Thus a principal is able to read (write) when she has—or can collude to acquire—a read (write) key for each seal. In the sequent, a DLM label will be translated into a message which discloses read and write keys according to the label's meaning. This will follow the definitions of $reads$ and $writes$ from Section 2.2.

The seal corresponding to $\pi = o : \overline{r} \,!\, \overline{w}$ is written $\mathrm{P}[\![\pi]\!]$ and is intended to ensure two properties. A envelope sealed with $\mathrm{P}[\![\pi]\!]$ should only be written to with a private key belonging to o or a member of $\overline{w}$ and read from by principals with a private key from $o, \overline{r}$. To compile π we first generate a two fresh key pairs (K_W^-, K_W^+) and (K_R^-, K_R^+). Messages packed in envelopes with this seal will be encrypted by K_R^+ and signed by K_W^-. Hence the key K_W^- will serve as a (necessary but not sufficient) capability for writing, and K_R^- will serve as a capability for reading. Seal creation encrypts K_W^- and K_R^- such that only principals from $o, \overline{w}$ and $o, \overline{r}$ respectively can read them. The public keys are displayed in the clear. Restricting access to K_W^- enforces the policy's write component; dually, restricting access to K_R^- enforces the read component. Lastly, a string describing the policy's structure is prepended, and the entire seal is signed by o. This signature ensures that the seal authentically describes o's policy. The policy translation is described by the following pair of equations, where the subscripted R and W parameters make key generation explicit:

$$\begin{aligned}\mathrm{P}[\![o : \overline{r} \,!\, \overline{w}]\!]_{R,W} = sign(o, [&\texttt{"o : }\overline{\texttt{r}}\texttt{ ! }\overline{\texttt{w}}\texttt{"}, (K_R^+, K_W^+)] \\ &+\!\!+ \ encFor\ (o, \overline{r})\ K_R^- \\ &+\!\!+ \ encFor\ (o, \overline{w})\ K_W^-)\end{aligned}$$

$$encFor\ (p_1 \ldots p_n)\ m = [enc(K_{p_1}^+, m) \ldots enc(K_{p_2}^+, m)].$$

A label comprises one or more ordered policies. We compile a label to an envelope by mapping the constituent poli-

$$\boxed{\sigma \vdash_d m}$$

$$\frac{(\mathit{knows}\ m) \in \sigma}{\sigma \vdash_d m}\ \textsc{d-Taut} \qquad \frac{\sigma \vdash_u K \qquad \sigma \vdash_d m}{\sigma \vdash_d \mathit{enc}(K, m)}\ \textsc{d-Encrypt} \qquad \frac{\sigma \vdash_d \mathit{enc}(K^+_\kappa, m) \qquad \sigma \vdash_u K^-_\kappa}{\sigma \vdash_d m}\ \textsc{d-Decrypt}$$

$$\frac{\sigma \vdash_u K \qquad \sigma \vdash_d m}{\sigma \vdash_d \mathit{sign}(K, m)}\ \textsc{d-Sign} \qquad \frac{\sigma \vdash_d \mathit{sign}(K, m)}{\sigma \vdash_d m}\ \textsc{d-Sign-Id} \qquad \frac{\sigma \vdash_b m}{\sigma \vdash_d m}\ \textsc{d-Lift} \qquad \frac{\sigma \vdash_d \mathit{enc}(K, m)}{\sigma \vdash_d K}\ \textsc{d-Enc-Id}$$

$$\frac{\sigma \vdash_d (m_1, m_2)}{\sigma \vdash_d m_1}\ \textsc{d-PairL} \qquad \frac{\sigma \vdash_d (m_1, m_2)}{\sigma \vdash_d m_2}\ \textsc{d-PairR} \qquad \frac{\sigma \vdash_d m_1 \qquad \sigma \vdash_d m_2}{\sigma \vdash_d (m_1, m_2)}\ \textsc{d-Pair}$$

$$\boxed{\sigma \vdash_u m} \qquad \boxed{\sigma \vdash_b m}$$

$$\overset{\textsc{u-Taut}}{\frac{(\mathit{actswith}\ m) \in \sigma}{\sigma \vdash_u m}} \qquad \overset{\textsc{u-Lift}}{\frac{\sigma \vdash_d m}{\sigma \vdash_u m}} \qquad \overset{\textsc{b-Taut}}{\frac{(\mathit{believes}\ m) \in \sigma}{\sigma \vdash_b m}} \qquad \overset{\textsc{b-Sign-Verify}}{\frac{\sigma \vdash_d \mathit{sign}(K^-, m) \qquad \sigma \vdash_b K^+}{\sigma \vdash_b m}}$$

Figure 7. Cryptographic Deduction System

cies list to to a list of seals.

$$\begin{aligned} &\mathrm{L}[\![(\pi_1 \ldots \pi_n)]\!]_{R1,W1,\ldots Rn,Wn} \\ &\quad = \quad [\mathrm{P}[\![\pi_1]\!]_{R1,W1} \ldots \mathrm{P}[\![\pi_n]\!]_{Rn,Wn}] \end{aligned}$$

Once ℓ is translated to an envelope, we can proceed with translating $\langle v \rangle_\ell$. First we recursively translate v to obtain $\mathrm{V}[\![v]\!]$. Next we write into the envelope by encrypting $\mathrm{V}[\![v]\!]$ with each seal's K^-_R and signing with K^-_W key. The result is paired with the list of seals. Formally, the value translation is given by

$$\begin{aligned} \mathrm{V}[\![i]\!]_{\overline{\kappa}} &= \texttt{"}i\texttt{"} \\ \mathrm{V}[\![\texttt{inl}\ v]\!]_{\overline{\kappa}} &= (\texttt{"inl"}, \mathrm{V}[\![v]\!]_{\overline{\kappa}}) \\ \mathrm{V}[\![\texttt{inr}\ v]\!]_{\overline{\kappa}} &= (\texttt{"inr"}, \mathrm{V}[\![v]\!]_{\overline{\kappa}}) \\ \mathrm{V}[\![\langle v \rangle_\ell]\!]_{\overline{\kappa}_1, \overline{\kappa}_2} &= (\mathit{doPack}\ \overline{\kappa}_2\ \mathrm{V}[\![v]\!]_{\overline{\kappa}_1}, \mathrm{L}[\![\ell]\!]_{\overline{\kappa}_2}) \end{aligned}$$

where

$$\begin{aligned} &\mathit{doPack}\ (R_1, W_1, \ldots, R_n, W_n)\ m \\ &\qquad\qquad = \quad \mathit{es}(R_n, W_n, \ldots \mathit{es}(R_1, W_1, m) \ldots) \\ &\mathit{es}(R, W, m) \quad = \quad \mathit{sign}(K^-_W, \mathit{enc}(K^+_R, m)). \end{aligned}$$

Entire memories are translated to cryptographic states by packing each location's contents:

$$\begin{aligned} \mathrm{M}[\![\cdot]\!]^{\Theta}_{\overline{\kappa}} &= \emptyset \\ \mathrm{M}[\![M[x \mapsto v]]\!]^{\Theta}_{\overline{\kappa}, \overline{\kappa}'} &= \mathrm{M}[\![M]\!]^{\Theta}_{\overline{\kappa}}, \mathit{knows}\ \mathrm{V}[\![\langle v \rangle_\ell]\!]_{\overline{\kappa}'} \\ &\qquad (\text{where } \Theta(x) = \tau\{\ell\}) \end{aligned}$$

To correctly thread $\overline{\kappa}$ though the above calculation, we assume that locations in a program are translated in a fixed order. Picking an order is easy, as memories are finite and the choice of order is arbitrary.

3.3 Cryptographic Analysis

We model the execution of a SImp program by an evolving cryptographic state that reflects the program's dynamic memory contents. To do this we will define equivalence relations on messages and cryptographic states, then show that the translation functions from Section 3.2 preserve equivalences. Lastly, we introduce state transition rules for cryptographic states and argue that command evaluation corresponds to state evolution.

First we connect the cryptographic system and our DLM. The predicate $\sigma \succ \overline{p}$ holds when $\forall p \in \overline{p}. \sigma \vdash_u K^-_p$. The relation $\sigma \vdash m \cong m'$ means that in state σ messages m and m' are indistinguishable. It is inductively defined by the rules in Figure 8. Rule ME-ID states that identical terms are indistinguishable. Rule ME-ENC-SECRET assumes perfect cryptographic operations. In particular, ME-ENC-SECRET states that if two messages cannot be decrypted, they are equivalent—you cannot provide evidence that they do not encrypt equivalent values. The rule ME-PAIR says that information from the left sides of pairs is considered when checking the right sides, and vice versa. This is necessary to avoid erroneously deriving that $(K, \mathit{enc}(K, \texttt{"a"})) \cong_\ell (K, \mathit{enc}(K', \texttt{"b"}))$.

Rule ME-SIGN-SUSPECT requires explanation. It states that two messages are equivalent when they are signed by untrusted keys. From the perspective of an honest user, this makes perfect sense. Honest players will ignore signed messages that they do not trust. It also makes sense from the perspective of the attacker. A Dolev-Yao attacker only desires to construct (or deconstruct) messages he should not be allowed to do. This aspect of signatures is handled by D-SIGN and D-SIGN-ID. While an attacker might swap two

$$\boxed{\sigma \vdash m_1 \cong m_2}$$

$$\frac{}{\sigma \vdash m \cong m}\ \textsc{me-Id}$$

$$\frac{\begin{array}{c}\sigma, \mathit{knows}\ m_2, \mathit{knows}\ m_2' \vdash m_1 \cong m_1' \\ \sigma, \mathit{knows}\ m_1, \mathit{knows}\ m_1' \vdash m_2 \cong m_2'\end{array}}{\sigma \vdash (m_1, m_2) \cong (m_1', m_2')}\ \textsc{me-Pair}$$

$$\frac{\sigma \vdash m_1 \cong m_2}{\sigma \vdash \mathit{enc}(K, m_1) \cong \mathit{enc}(K, m_2)}\ \textsc{me-Enc-Struct}$$

$$\frac{\sigma \not\vdash_u K_1^- \qquad \sigma \not\vdash_u K_2^-}{\sigma \vdash \mathit{enc}(K_1^+, m_1) \cong \mathit{enc}(K_2^+, m_2)}\ \textsc{me-Enc-Secret}$$

$$\frac{\sigma \vdash m_1 \cong m_2}{\sigma \vdash \mathit{sign}(K, m_1) \cong \mathit{sign}(K, m_2)}\ \textsc{me-Sign-Struct}$$

$$\frac{\sigma \not\vdash_b K_1^+ \qquad \sigma \not\vdash_b K_2^+}{\sigma \vdash \mathit{sign}(K_1^-, m_1) \cong \mathit{sign}(K_2^-, m_2)}\ \textsc{me-Sign-Suspect}$$

Figure 8. Contextual Message Equivalence

equivalent messages, he will only be able to fool an honest participant who does not believe the keys used to sign the messages; this is harmless.

The next four definitions extend the notion of ℓ-equivalence to messages.

Definition 3 (σ *reads* ℓ)**.**
If $\exists \overline{p}.(\overline{p}\ \mathit{reads}\ \ell \wedge \forall p \in \overline{p}. \sigma \vdash_u K_p^-)$ *then* σ *reads* ℓ.

Definition 4 (σ *distrusts* ℓ)**.**
If $\exists \overline{p}.(\overline{p}\ \mathit{writes}\ \ell \wedge \forall p \in \overline{p}. \sigma \not\vdash_b K_p^-)$ *then* σ *distrusts* ℓ.

Definition 5 ($\sigma \leq \ell$)**.**
If $\neg\sigma$ *reads* ℓ *and* σ *distrusts* ℓ *then* $\sigma \leq \ell$.

Definition 6 (Message ℓ-equivalence: $m \cong_\ell m'$)**.**
We write $m \cong_\ell m'$ *if and only if*

$$\forall \sigma.\ \sigma \leq \ell \implies \sigma, \mathit{knows}\ m, \mathit{knows}\ m' \vdash m \cong m'$$

For example, assume K_R^- is fresh, then consider the messages

$$\begin{aligned} m &= (\mathit{enc}(K_p^+, K_R^-), \mathit{enc}(K_R^+, \texttt{"3"})) \\ m' &= (\mathit{enc}(K_p^+, K_R^-), \mathit{enc}(K_R^+, \texttt{"4"})) \end{aligned}$$

and the label $\ell = \{o : pq\,!\,\mathcal{P}\}$. Is it true that $m \cong_\ell m'$? To find out, we take arbitrary σ where $\sigma \leq \ell$. Unfolding Definition 5 shows $\neg(\sigma\ \mathit{reads}\ \ell)$. Therefore, because $p\ \mathit{reads}\ \ell$, we conclude $\sigma \not\vdash_u K_p^-$. Now let

$$\begin{aligned} \sigma' &= \sigma, \mathit{knows}\ (\mathit{enc}(K_p^+, K_R^-)) \\ \sigma'' &= \sigma', \mathit{knows}\ m, \mathit{knows}\ m'. \end{aligned}$$

$$\boxed{\vdash \sigma \rightarrow \sigma'}$$

$$\frac{}{\vdash \sigma \rightarrow \sigma, \mathit{knows}\ (K_\kappa^+, K_\kappa^-)}\ \textsc{cs-Fresh}\quad \kappa \text{ fresh}$$

$$\textsc{cs-Derive}\ \frac{\sigma \vdash_d m}{\vdash \sigma \rightarrow \sigma, \mathit{knows}\ m} \qquad \textsc{cs-Forget}\ \frac{\sigma' \subseteq \sigma}{\vdash \sigma \rightarrow \sigma'}$$

$$\textsc{cs-Compute}\ \frac{\sigma \vdash_d \texttt{"i}_1\texttt{"} \qquad \sigma \vdash_d \texttt{"i}_2\texttt{"}}{\vdash \sigma \rightarrow \sigma, \texttt{"i}_3\texttt{"}}\ \text{where } i_3 = i_1 + i_2$$

Figure 9. Cryptographic State Transitions

From $\sigma \not\vdash_u K_p^-$ and the freshness of K_R^-, it is clear that $\sigma' \not\vdash_d K_R^-$ and that $\sigma'' \not\vdash_d K_R^-$. Therefore $\sigma'' \vdash \mathit{enc}(K_R^+, \texttt{"3"}) \cong \mathit{enc}(K_R^+, \texttt{"4"})$ by ME-ENC-SECRET. Applying ME-PAIR yields $\sigma' \vdash m \cong m'$. Finally generalizing over σ and applying Definition 6 gives $m \cong_\ell m'$.

The messages m and m' contain the key components of $\mathrm{V}[\![\langle 3\rangle_{\{o:p\,!\,\}}]\!]$ and $\mathrm{V}[\![\langle 4\rangle_{\{o:p\,!\,\}}]\!]$. Demonstrating their ℓ-equivalence here is intended to be suggestive, and this relation will be made precise by Lemma 3. First, however, we must lift ℓ-equivalence to cryptographic states.

Definition 7 ($\sigma \cong_\ell \sigma'$)**.** *If knows* $m \in \sigma$ *implies that there exists* m' *where knows* $m' \in \sigma'$ *and* $m \cong_\ell m'$*, then* $\sigma' \vDash_\ell \sigma$*. If* $\sigma' \vDash_\ell \sigma$ *and* $\sigma \vDash_\ell \sigma'$*, then* $\sigma \cong_\ell \sigma'$.

Thus far we have defined two sorts of equivalence relations: those at the SImp level and those at the cryptographic level. However, it is not yet clear what, if any, formal relation exists between, say, value and message equivalences. For our cryptographic semantics to provide a safe interpretation of of SImp, equivalent values (memories) must translate to equivalent messages (states). Otherwise, an attacker could illegally gain information by observing the cryptographic states corresponding to the beginning and end of a well-typed program's execution. The following lemma and corollary state that this cannot occur.

Lemma 3 (Adequacy of Value Translation)**.** *If* $v_1 \cong_\ell v_2$ *and* $\overline{\kappa}$ *is fresh then* $\mathrm{V}[\![v_1]\!]_{\overline{\kappa}} \cong_\ell \mathrm{V}[\![v_2]\!]_{\overline{\kappa}}$.

Corollary 1 (Adequacy of Memory Translation)**.** *If* $\Theta \vdash M_1 \cong_\ell M_2$ *and* $\overline{\kappa}$ *is fresh then* $\mathrm{M}[\![M_1]\!]_{\overline{\kappa}}^{\Theta} \cong_\ell \mathrm{M}[\![M_2]\!]_{\overline{\kappa}}^{\Theta}$.

Coupled with Theorem 2, these demonstrate that our cryptographic system reflects language level noninterference. Thus Lemma 3 and its corollary are the statements that our cryptographic system is safe.

Above, we showed that SImp and its cryptographic interpretation are safe. However, safety and implementability are orthogonal issues. For example, the translation that maps all values to the empty string is safe, but could not be the basis for practical system. We argue that our cryptographic model is a reasonable foundation for SImp in two steps. First, we define a nondeterministic transition relation on cryptographic states in which only cryptographically realizable transitions may occur. Second, we claim that it simulates SImp evaluation.

Intuitively, the relation $\vdash \sigma_1 \rightarrow^* \sigma_2$ holds when a cryptographic state σ_1 can, using basic cryptographic operations, transition to state σ_2. This is the reflexive transitive closure of the rules given in Figure 9. We are most interested in states corresponding to memories (i.e. heaps), and those corresponding to expressions currently executing (i.e. stacks). State $state(\overline{\kappa}, \overline{p}, c)$ represents the dynamic information associated with command c and principals $\overline{p}$. It is defined by

$$\begin{aligned}
\sigma_0 &= \{knows\ \texttt{"inl"}, knows\ \texttt{"inr"}, \ldots\} \\
&\quad \cup \{knows\ K_p^+ \mid p \in \mathcal{P}\} \\
\sigma_{\overline{\kappa}}^c &= \{\mathrm{V}[\![v_i]\!]_{\overline{\kappa}_i} \mid \ldots v_i \ldots = values(c)\} \\
&\quad \cup \{\mathrm{V}[\![\ell_i]\!]_{\overline{\kappa}_i} \mid \ldots \ell_i \ldots = labels(c)\} \\
state(\overline{\kappa}, \overline{p}, c) &= \sigma_0 \cup \sigma_{\overline{\kappa}}^c \cup \{actswith\ K_p^- \mid p \in \overline{p}\}
\end{aligned}$$

where $values(c)$ is the list of values occurring in command c and $labels(c)$ is the list of labels occurring in c. Note that because $\sigma_{\overline{\kappa}}^c$ contains translations of labels occurring in source program c, all owners' keys are needed at compile time (but, of course, not later). Such a requirement is also intuitively necessary; after all c must be treated as very high integrity data.

Theorem 3 (Feasibility)**.** *If* $\Theta \vdash M\ OK$, $pc; \Theta; \Gamma \vdash c$, $\overline{p}\ reads\ pc$, $\overline{p}\ writes\ pc$, *and* $\overline{p} \vdash \langle M, c\rangle \rightarrow \langle M', c'\rangle$ *then* $\exists \overline{\kappa}_3, \overline{\kappa}_4.\ \vdash \mathrm{M}[\![M]\!]_{\overline{\kappa}_1}^{\Theta} \cup state(\overline{\kappa}_2, \overline{p}, c) \rightarrow^* \mathrm{M}[\![M']\!]_{\overline{\kappa}_3}^{\Theta} \cup state(\overline{\kappa}_4, \overline{p}, c)$.

The cryptographic semantics's non-determinism allows us to investigate feasibility without picking a particular implementation strategy and providing a fully-abstract simulation of SImp programs. Thus Theorem 3, demonstrates that there is *some* cryptographic realization of memory transitions described by the program. However, it does not need to reflect all the computational detail of the program's operation (e.g. maintenance of the run-time stack) into the cryptographic transition system.

The commutation diagram below summarizes our main results. For convenience, we write $\overline{p}; c; \overline{\kappa} \vdash \sigma \rightarrow^* \sigma'$ for $\vdash state(\overline{\kappa}, \overline{p}, c) \cup \sigma \rightarrow^* \sigma'$. The diagram's inner and outer loops each illustrate Theorem 3. The preservation of ℓ-equivalence by the top and side arrows demonstrates Theorem 2 and Corollary 1.

$$\begin{tikzcd}
M_1 \arrow[r, phantom, "\cong_\ell"] \arrow[rrr, bend left, "\overline{p} \vdash \langle \_, c\rangle \rightarrow^* \langle \_, \texttt{skip}\rangle"] \arrow[d, "\mathrm{M}[\![\cdot]\!]_{\overline{\kappa}}^{\Theta}"'] & M_2 \arrow[r, "\overline{p} \vdash \langle \_, c\rangle \rightarrow^* \langle \_, \texttt{skip}\rangle"'] \arrow[d, "\mathrm{M}[\![\cdot]\!]_{\overline{\kappa}}^{\Theta}"] & M_2' \arrow[r, phantom, "\cong_\ell"] \arrow[d, dashed, "\mathrm{M}[\![\cdot]\!]_{\overline{\kappa}}^{\Theta}"'] & M_1' \arrow[d, dashed, "\mathrm{M}[\![\cdot]\!]_{\overline{\kappa}}^{\Theta}"] \\
\sigma_1 \arrow[r, phantom, "\cong_\ell"] \arrow[rrr, dashed, bend right, "\overline{p}; c; \overline{\kappa} \vdash \_ \rightarrow^* \_"'] & \sigma_2 \arrow[r, dashed, "\overline{p}; c; \overline{\kappa} \vdash \_ \rightarrow^* \_"] & \sigma_2' \arrow[r, phantom, "\cong_\ell"] & \sigma_1'
\end{tikzcd}$$

We assume all memories and commands are appropriately typed with location context Θ, the empty value context, and a pc such that $\overline{p}\ reads\ pc$ and $\overline{p}\ writes\ pc$. To interpret the diagram, begin with ℓ-equivalent memories, M_1 and M_2. The arcs across the top correspond to terminating evaluations of command c. Theorem 2 shows $\Theta \vdash M_1' \cong_\ell M_2'$. The arrows going down are memory translations. Corollary 1 shows $\sigma_1 \cong_\ell \sigma_2$ and $\sigma_1' \cong_\ell \sigma_2'$; this reflects safety. Lastly, the arcs across the bottom illustrate that by, Theorem 3, the system transformations of the program are feasible. The right hand side arrows point down because that is sufficient to demonstrate feasibility. Reversing these arrows up would require a fully-abstract translation of SImp to the cryptographic semantics—an interesting problem outside the scope of this paper.

4 Related Work

Askarov, Hedin, and Sabelfeld [4] recently investigated a type system for programs with encryption and with the property that all well typed programs are non-interfering. Their work differs from ours in several ways. They treat encryption, decryption, and key generation as language primitives. In contrast, we use cryptography implicitly to implement high-level language features. Askarov's language appears superior for modeling cryptographic protocols, and ours provides a cleaner and simpler interface for applications programming. The central technical difference is that Askarov and colleagues ensure noninterference completely by way of static checks; our noninterference result stems from the harmonious interplay of static and dynamic checking. Further comparison of the approaches is warranted.

Chothia, Duggan, and Vitek [12] examine a combination of DLM-style policies and cryptography, called the Key-Based DLM (KDLM). Their system, like Askarov's, provides an extensive set of language level cryptographic primitives and types inhabited by keys. In contrast to SImp, KDLM security typing is nominal—labels have names and each name corresponds to a unique cryptographic key. While they prove type soundness, Chothia and colleagues do not provide more specific security theorems such as non-interference.

Our $pack/unpack$ language feature can be compared with both dynamic types [2] and standard existential types [24]. Like typecase, `unpack` may fail at runtime; the standard existential unpack always succeeds. As with dynamic/typecase, our `pack/unpack` does not provide direct support for abstract datatypes; existentials usually do. A more refined approach to `pack/unpack` might use type annotation to expose the internal structure of encrypted values; this would resemble a existential package with a bounded type variable.

Sumii and Pierce [31] studied λ_{seal}, an extension to lambda calculus with terms of form $\{e\}_{e'}$, meaning e sealed-by e', and a corresponding elimination form. Like Askarov and colleagues, they make seal (i.e. key) generation explicit in program text; however their dynamic semantics, which include runtime checking of seals, is simpler than Askarov's. λ_{seal} includes black-box functions that analyze sealed values, but cannot be disassembled to reveal the seal (key). It is not clear how to interpret such functions in a cryptographic model.

Heintze and Riecke's SLam calculus [19] is an information flow lambda calculus in which the right to read a closure corresponds to the right to apply it. This sidesteps the black-box function issue from λ_{seal}. In SLam, some expressions are marked with the authority of the function writer. The annotations control declassification, and, we conjecture, are analogous to the pretranslated labels in SImp. Additionally SLam types have a nested form where, for example, the elements in a list and the list itself may be given different security annotations. Combined with `pack`, such nesting could facilitate defining data structures with dynamic and heterogeneous security properties.

We use the algebraic Dolev-Yao model to study the connection connection between information flow and cryptography. Laud and Vene [21] examined this problem using a computational model of encryption. More recently, Smith and Alpízar extended this work to include a model of decryption [30]. They prove noninterference for a simple language without declassification (or packing) and a two-point security lattice. Like Chothia and colleagues, they map labels to fixed keys.

Abadi and Rogaway proved that Dolev-Yao analysis is sound with respect to computational cryptographic analysis in a setting similar to our own [3]. While the inference system in Figure 7 was influenced by their formalism, there are significant differences in approach. In particular, Abadi and Rogaway do not discuss public key cryptography, which we use extensively. Backes and Pfitzmann [6] with Waidner [7] have also investigated the connection between symbolic and computational models of encryption. They define a Dolev-Yao style library and show that protocols proved secure with respect to library semantics are also secure with respect to computational cryptographic analysis. This library might provide an excellent foundation for further rigorous analysis of SImp.

5 Discussion

Information flow languages often provide escape hatches to *declassify* secrets or *endorse* untrusted input. While these mechanisms allow violations of language policies, they isolate locations where leaks can occur and are quite useful in practice. Unfortunately, languages with unrestricted declassification and endorsement no longer enjoy simple non-interference, leading to complex metatheory [10]. SImp's `pack` and `unpack` operators provide a middle ground. Like declassify, `pack` lowers confidentiality policies, and, like endorse, `unpack` lowers integrity label policies. However, packing and unpacking are not as general as declassification and endorsement. For example, pack/unpack cannot be used to make public the result of a password check—a classic use of declassification. The advantage of `pack/unpack`, is that they preserve non-interference and are thus safer than declassify/endorse. Thus these constructs are complementary. We believe a practical information flow language could include both.

5.1 Comparison with other DLMs

Several decentralized label models are discussed in the literature. As originally formulated by Myers and Liskov, structurally defined labels described only confidentiality (or, dually, integrity) policies [22]. Later research treated confidentiality and integrity simultaneously. Zdancewic and Tse examined a DLM where integrity polices define a "trusted by" relation [32]. In contrast, Myers and Chong treat integrity as we do, with the "written by" interpretation [10]. Lastly, Chothia, Duggan, and Vitek's KDLM blends structural and nominal label semantics [12].

Our DLM differs significantly from Myers and Liskov's original presentation [22], which gives labels a more restrictive interpretation. For example, in our setting label

$$\ell = \{\text{Alice} : \text{Alice}, \text{Charlie} \,!\, \emptyset; \text{Bob} : \text{Bob}, \text{Charlie} \,!\, \emptyset\}$$

can be read with the authority of $\{\text{Alice}, \text{Bob}\}$ or just Charlie. Myers requires Charlie's authority to read ℓ. (Of course, Alice and Bob may conspire to first declassify and then read—but it's important not to conflate this with simple reading.) Our choice of interpretation was motivated by the constraints inherent in cryptographically translating packages. In particular, Lemma 3 would not hold under Myers and Liskov's interpretation. However, we could retain this result by changing the definition of $\mathrm{v}[\![\cdot]\!]_.$ to use *share semantics*. Under share semantics, $\langle v\rangle_\ell$ is translated by generating a fresh key pair and encrypting v with the public key. Cryptographic shares of the fresh keys are distributed according to each owner's read and write policy. With mutual distrust among owners, no owner should be able to learn the fresh keys except as permitted by the $reads$ and $writes$ relations. This requires generating key shares without revealing the underlying keys. We hoped to do so with threshold cryptography [15], but current approaches expose one key of the pair.

Previous DLMs include a partial order on principals called the *acts-for hierarchy* [22]. If $p \succ q$ then principal p is assumed to have the authority of q. If σ is p's cryptographic state, $\sigma \vdash_u K_q^-$ models this acts-for relation in our setting. This is a course-grained form of delegation. The correct cryptographic implementation of the acts for hierarchy is not clear. A naive implementation might provide Alice with Bob's private key when Alice $\succ$ Bob. However this has practical shortcomings: revocation is impossible, and Bob cannot selectively grant Alice rights. A more sophisticated protocol might require that Bob provide Alice with a network service or a smart card that can selectively provide encryption and signing services.

5.2 Language Extensions

SImp is a core language for programming with information flow and packing. Future work may extend it with several new constructs.

Currently SImp programs must unpack packages to compute with their contents. Alternatively, the rule

$$\frac{\overline{p}; M \vdash v_1 + v_2 \to v_3}{\overline{p}; M \vdash \langle v_1\rangle_\ell + \langle v_2\rangle_\ell \to \langle v_3\rangle_\ell}\ \textsc{Bind}$$

would permit computation within packages. (The name BIND follows the monadic interpretation of security statements in Abadi's Dependency Core Calculus [1].) While BIND can be implemented using the homomorphic properties of the Goldwasser-Micali cryptosystems, they cannot sustain an analogous multiplication rule. Other systems (e.g. RSA) would support multiplication but not addition. Unfortunately, current cryptosystems can only provide efficient homomorphic computation over a single algebraic operator. A more general bind would require an efficient homomorphic algebraic (i.e. additive and multiplicative) cryptosystem; the existence of such schemes is an open problem [26].

Imperative update of packed values is compatible with SImp's semantics. The operational semantics might look like

$$\frac{\overline{p}\ writes\ \ell}{\overline{p}; M \vdash \texttt{put}\ v\ \texttt{in}\ \langle v_0\rangle_\ell \to \texttt{inl}\ \langle v\rangle_\ell}\ \textsc{Put}\ .$$

PUT assigns a low (trusted, public) value into a high (tainted, secret) package; this is straightforward to typecheck and dynamically safe. What distinguishes packing and writing? Compiling a `pack` requires the creation of a new envelope, which in turn requires the owners' keys. In contrast, a `put` reuses dynamically acquired envelopes and requires no compile-time keys. Our model assumes that programs are compiled with the authority of all owners; thus PUT conveys no particular advantage. It may be useful under weaker assumptions, such as those encountered in the context of program partitioning [35].

Lastly, SImp could allow programs to strengthen the label of a (potentially unreadable) package. In full generality,

$$\frac{\ell_1 \le \ell_2}{\overline{p}; M \vdash \texttt{strengthen}\ \langle v\rangle_{\ell_1}\ \texttt{to}\ \ell_2 \to \langle v\rangle_{\ell_2}}\ \textsc{Strengthen}\ .$$

In the case of a DLM, ℓ_2 may be more restrictive than ℓ_1 in two ways: ℓ_2 may have new that policies ℓ_1 does not, or ℓ_2 may have more restrictive policies than ℓ_1. In the former case, it is straightforward to append the new policy's seal to ℓ_1's envelope and finish the construction of $\mathrm{v}[\![\langle v\rangle_{\ell_2}]\!]$. However, it is not clear what to do in the second case.

Conclusion It is important to consider the interplay between cryptography and information-flow in the context of language-based security. This paper has investigated one design for high-level language features that make it easy for to connect a secure program's confidentiality and integrity labels with an underlying cryptographic implementation. Our package mechanism complements existing, more general approaches to downgrading, but has the advantage of yielding a strong noninterference result against Dolev-Yao attackers. We expect that such packages will be useful for building systems that enforce strong security policies, even when confidential data must leave the confines of the trusted run-time environment.

Acknowledgments We would like to thank the anonymous reviewers for their helpful comments and Peeter Laud for his suggestions regarding the cryptographic semantics.

References

[1] M. Abadi. Access control in a core calculus of dependency. In *ICFP '06: Proceedings of the eleventh ACM SIGPLAN international conference on Functional Programming*, pages 263–273, Portland, Oregon, USA, September 2006.

[2] M. Abadi, L. Cardelli, B. Pierce, and D. Rémy. Dynamic typing in polymorphic languages. *Journal of Functional Programming*, 5(1):111–130, January 1995.

[3] M. Abadi and P. Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). *Journal of Cryptology*, 15(2):103–127, 2002.

[4] A. Askarov, D. Hedin, and A. Sabelfeld. Cryptographically masked information flows. In *Proceedings of the International Static Analysis Symposium*, LNCS, Seoul, Korea, August 2006.

[5] A. Askarov and A. Sabelfeld. Security-typed languages for implementation of cryptographic protocols: A case study. In *Proceedings of the 10th European Symposium on Research in Computer Security (ESORICS)*, Milan, Italy, September 2005.

[6] M. Backes and B. Pfitzmann. Relating symbolic and cryptographic secrecy. *IEEE Trans. Dependable Secur. Comput.*, 2(2):109–123, 2005.

[7] M. Backes, B. Pfitzmann, and M. Waidner. A composable cryptographic library with nested operations. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 220–230, Washington D.C., USA, 2003. ACM Press.

[8] D. E. Bell and L. J. LaPadula. Secure computer system: Unified exposition and Multics interpretation. Technical Report ESD-TR-75-306, MITRE Corp. MTR-2997, Bedford, MA, 1975. Available as NTIS AD-A023 588.

[9] R. Chapman. Industrial experience with spark. *Ada Lett.*, XX(4):64–68, 2000.

[10] S. Chong and A. C. Myers. Decentralized robustness. In *Proceedings of the 19th IEEE Computer Security Foundations Workshop (CSFW'06)*, pages 242–253, Los Alamitos, CA, USA, July 2006.

[11] S. Chong, A. C. Myers, K. Vikram, and L. Zheng. *Jif Reference Manual*, June 2006. Available from http://www.cs.cornell.edu/jif.

[12] T. Chothia, D. Duggan, and J. Vitek. Type based distributed access control. In *Proceedings of the 16th IEEE Computer Security Foundations Workshop (CSFW'03)*, Asilomar, Ca., USA, July 2003.

[13] D. E. Denning. *Secure Information Flow in Computer Systems*. PhD thesis, Purdue University, W. Lafayette, Indiana, USA, May 1975.

[14] D. E. Denning and P. J. Denning. Certification of Programs for Secure Information Flow. *Comm. of the ACM*, 20(7):504–513, July 1977.

[15] Y. G. Desmedt and Y. Frankel. Threshold cryptosystems. In *CRYPTO '89: Proceedings on Advances in cryptology*, pages 307–315, New York, NY, USA, 1989. Springer-Verlag New York, Inc.

[16] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 2(29), 1983.

[17] D. Duggan. Cryptographic types. In *CSFW '02: Proceedings of the 15th IEEE Computer Security Foundations Workshop (CSFW'02)*, page 238, Washington, DC, USA, 2002. IEEE Computer Society.

[18] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society Press, Apr. 1982.

[19] N. Heintze and J. G. Riecke. The SLam calculus: programming with secrecy and integrity. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 365–377, New York, NY, USA, 1998. ACM Press.

[20] B. Hicks, K. Ahmadizadeh, and P. McDaniel. Understanding practical application development in security-typed languages. In *22st Annual Computer Security Applications Conference (ACSAC)*, Miami, Fl, December 2006.

[21] P. Laud and V. Vene. A type system for computationally secure information flow. In *Proceedings of the 15th International Symposium on Fundamentals of Computational Theory*, volume 3623, pages 365–377, Lübeck, Germany, 2005.

[22] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, 2000.

[23] A. C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification and qualified robustness. *Journal of Computer Security*, 2006. To appear.

[24] B. C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, Massachusetts, 2002.

[25] F. Pottier and V. Simonet. Information flow inference for ML. In *Proc. 29th ACM Symp. on Principles of Programming Languages (POPL)*, pages 319 – 330, Portland, Oregon, Jan. 2002.

[26] D. K. Rappe. *Homomorphic Cryptosystems and Their Applications*. PhD thesis, University of Dortmund, Germany, 2004.

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

[28] Security-enhanced Linux. Project website `http://www.nsa.gov/selinux/` accessed November, 2006.

[29] V. Simonet. Flow Caml in a nutshell. In G. Hutton, editor, *Proceedings of the first APPSEM-II workshop*, pages 152–165, Mar. 2003.

[30] G. Smith and R. Alpízar. Secure information flow with random assignment and encryption. In *Proceedings of The 4th ACM Workshop on Formal Methods in Security Engineering: From Specifications to Code (FSME'06)*, pages 33–43, Alexandria, Virgina, USA, November 2006.

[31] E. Sumii and B. C. Pierce. A bisimulation for dynamic sealing. In *Principals of Programming Languages*, Venice, Italy, January 2004.

[32] S. Tse and S. Zdancewic. Run-time principals in information-flow type systems. In *IEEE Symposium on Security and Privacy*, 2004.

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

[34] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, Cambridge, Massachusetts, 1993.

[35] L. Zheng, S. Chong, S. Zdancewic, and A. C. Myers. Building Secure Distributed Systems Using Replication and Partitioning. In *IEEE 2003 Symposium on Security and Privacy*. IEEE Computer Society Press, 2003.