Cryptographically-Masked Flows

Aslan Askarov   Daniel Hedin   Andrei Sabelfeld

Department of Computer Science and Engineering
Chalmers University of Technology
412 96 Göteborg, Sweden

Abstract

Cryptographic operations are essential for many security-critical systems. Reasoning about information flow in such systems is challenging because typical (noninterference-based) information-flow definitions allow no flow from secret to public data. Unfortunately, this implies that programs with encryption are ruled out because encrypted output depends on secret inputs: the plaintext and the key. However, it is desirable to allow flows arising from encryption with secret keys provided that the underlying cryptographic algorithm is strong enough. In this article we conservatively extend the noninterference definition to allow safe encryption, decryption, and key generation. To illustrate the usefulness of this approach, we propose (and implement) a type system that guarantees noninterference for a small imperative language with primitive cryptographic operations. The type system prevents dangerous program behavior (e.g., giving away a secret key or confusing keys and non-keys), which we exemplify with secure implementations of cryptographic protocols. Because the model is based on a standard noninterference property, it allows us to develop some natural extensions. In particular, we consider public-key cryptography and integrity, which accommodate reasoning about primitives that are vulnerable to chosen-ciphertext attacks.

1 Introduction

Cryptographic operations are ubiquitous in security-critical systems. Reasoning about information flow in such systems is challenging because typical information-flow definitions allow no flow from secret to public data. The latter requirement underlies *noninterference* [12,17], which demands that public outputs are unchanged as secret inputs are varied. While traditional noninterference breaks in the presence of cryptographic operations, the challenge is to distinguish between breaking noninterference because of legitimate use of sufficiently strong encryption and breaking noninterference due to an unintended leak.

A common approach to handling cryptographic primitives in information-flow aware systems is by allowing *declassification* of encryption results. The intention of declassification is that the result of encryption can be released to the attacker. Declassification, however, is a versatile mechanism: different declassification dimensions correspond to different reasons why information is released [32,3]. Attempts at framing cryptographically-masked flows into different dimensions have been made although, as we discuss, not always with satisfactory results.

In this article, we introduce cryptographic primitives into an information-flow setting while preserving a form of noninterference property. This is achieved by building into the model a basic assumption that attackers may not distinguish between ciphertexts and that decryption using the wrong key fails. Although this assumption is stronger than some probabilistic and computational cryptographic models (which allow some information to leak when comparing ciphertexts), we argue that it can still be reasonable, and that it opens up possibilities for tracking information flow in the presence of cryptographic primitives in expressive programming languages.

The intuition behind our approach is sketched below and illustrated in Figure 1, where dashed and solid lines correspond to secret and public values, respectively. Fixing some public (low) input z_L and varying secret (high) input from x_H to y_H may not reflect on a public output z'_L of a system that satisfies noninterference (illustrated in Figure 1(a)). Suppose the system in question involves encryption, such as in the program $z = \texttt{encrypt}(k, x)$ for some secret key k. Clearly, noninterference is broken: variation in the secret input from x_H to y_H may cause variation in the public output from z'_L to z''_L (illustrated in Figure 1(b)).

However, noninterference can be recovered if the result of encryption is possibly *any* value v. This means that variation of the high input from x_H to y_H does not affect the public output—any value v is a possible public output in both cases. This form of noninterference is known as *possibilistic noninterference* [25] (illustrated in Figure 1(c)). Overall, although low outputs might depend on low inputs and ciphertexts, no observation about possible low outputs may reveal information about changes in high inputs (illustrated in Figure 1(d)).

This article makes a case for possibilistic noninterference as a natural model for cryptographically-masked flows. We show that a naive approach of collapsing all ciphertexts as indistinguishable opens up possibilities for *occlusion* [32], where masking an intended information flow in an indistinguishability definition may also mask other unintended leaks. Therefore, we propose a finer indistinguishability relation that not only avoids occlusion but also, by a recent result by Laud [22], guarantees computational security under some natural assumptions on the cryptographic primitives. With such a result at hand, our model allows focusing on enforcing a simple possibilistic property, which comes with a computational guarantee "for free."

To demonstrate that enforcing possibilistic noninterference is straightforward, we

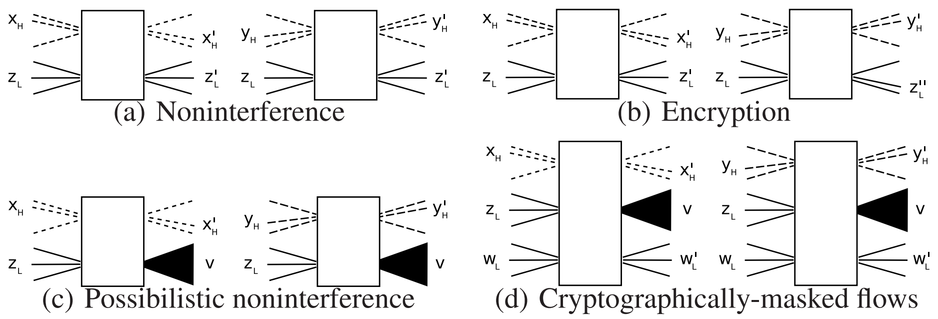
x_H
z_L
x'_H
z'_L
y_H
y'_H
(a) Noninterference
(b) Encryption
z''_L
(c) Possibilistic noninterference
v
(d) Cryptographically-masked flows
w_L
w'_L

Fig. 1. From noninterference to cryptographically-masked flows

have designed and implemented a security type system that provably enforces possibilistic noninterference for an imperative language with primitive cryptographic operations and communication channels. The type system prevents dangerous program behavior (e.g., giving away a secret key or confusing keys or non-keys), which we exemplify with secure implementations of cryptographic protocols. Because the model is based on a standard noninterference property, it allows us to develop some natural extensions. In particular, we consider public-key cryptography and integrity, which accommodates reasoning about primitives that are vulnerable to chosen-ciphertext attacks. The main soundness result (that the type system indeed guarantees security) is based on our formalization in the proof assistant Coq.

This article is a revised and extended version of [2]. Compared to the earlier version, the most significant contribution is a formalization of the soundness proof in the proof assistant Coq, which has helped crystallizing definitions and making our assumptions about cryptographic schemes more precise. In addition to this we have modified the language to obtain a clearer semantics, included the full presentation of the semantic and typing rules, expanded the wide-mouthed frog protocol example, updated the related work, and made other improvements throughout the article.

2 Language

We explore how to model cryptographic flows in a small imperative language equipped with encryption and decryption primitives, dynamic key generation, and channels for communication. This section introduces the syntax and semantics of the language.

2.1 Syntax

The syntax of the language is defined in Figure 2. Let $x \in \mathit{VarName}$ range over the set of variable names, $ch \in \mathit{ChanName}$ range over the set of channel names, A

$$\begin{array}{llll@{\qquad}llll}
\text{sec. levels} & \sigma & ::= & \mathtt{L} \mid \mathtt{H} & \text{basic types} & t & ::= & \mathtt{int} \mid \mathtt{enc}_{\gamma}\ \tau \\
\text{key levels} & \gamma & ::= & \mathtt{LK} \mid \mathtt{HK} & \text{prim. types} & \tau & ::= & t\ \sigma \mid \mathtt{key}\ \gamma \mid (\tau_1, \tau_2)
\end{array}$$

$$\begin{array}{llll}
\text{expressions} & e & ::= & n \mid x \mid e_1\ op\ e_2 \mid \mathtt{encrypt}_{\gamma}\ (e_1, e_2) \mid \mathtt{decrypt}_{\gamma}\ (e_1, e_2) \\
 & & \mid & (e_1, e_2) \mid \mathtt{fst}(e) \mid \mathtt{snd}(e) \\
\text{statements} & c & ::= & \mathtt{skip} \mid x := e \mid c_1; c_2 \mid \mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2 \\
 & & \mid & \mathtt{while}\ e\ \mathtt{do}\ c \mid \mathtt{out}(ch, e) \mid \mathtt{in}(x, ch) \mid \mathtt{newkey}(x, \gamma) \\
\text{program} & p & ::= & A_1\ c_1 \ldots A_n\ c_n
\end{array}$$

Fig. 2. Syntax

range over the set of *actor names*, and let $n \in \mathbb{Z}$ range over the integers.

A *program* consists of a sequence of *actors*, where an actor is a named command. The key levels, ranged over by γ, declare the maximum value security level the key can safely encrypt. Intuitively, high keys are allowed to encrypt both secrets and non-secrets, whereas low keys are only allowed to encrypt non-secrets; a more thorough discussion of this is found in the section on semantics below. The commands include the standard commands of an imperative language, commands for sending on and receiving from a given channel, and a command for generating new keys. Apart from expressions for encryption and decryption, expressions are standard: integers, variables, total binary operators, pair formation, and projection. Key values and encrypted values are excluded from the expression syntax; from a security perspective neither should appear as constants in the program text.

2.2 Semantics

The semantics of the system is defined as a big-step operational semantics. The actors of a program run concurrently and interact with each other by sending and receiving messages on channels. We refrain from modeling the semantics for the entire system and instead provide semantics for isolated actors. Thus we deliberately ignore information flows via races and other flows that may arise in concurrent systems (cf. [30]). These flows are typically harder to exploit, although reasoning about such flows in our setting is still a worthwhile topic for future work.

We begin by defining the values of the language, which are used in the definition of the semantics for expressions and commands.

2.2.1 Values

Let $Key_{\mathtt{LK}}$ and $Key_{\mathtt{HK}}$ be two disjoint sets of keys, ranged over by $k_{\mathtt{LK}}$ and $k_{\mathtt{HK}}$ respectively, and let k range over $Key_{\mathtt{LK}} \cup Key_{\mathtt{HK}}$. Let $u \in U$ range over a set of bit strings representing the encrypted values. The *values* are built up by the *ordinary values*, integers, keys and pairs of values, together with the encrypted values.

$$\text{values} \in \textit{Value} \quad v ::= n \mid k_{\mathtt{LK}} \mid k_{\mathtt{HK}} \mid (v_1, v_2) \mid u$$

The system is parameterized over two *symmetric encryption schemes*—one for each key level γ. An encryption scheme is a triple $\mathcal{S} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$, with the following properties:

- $\mathcal{K}$ is a *key generation* algorithm that on each invocation generates a new key from the set Key_{γ} associated with the encryption scheme.
- $\mathcal{E}$ is a nondeterministic encryption algorithm that takes a key and a bit string and returns a bit string—the ciphertext. In the following, we use $\mathcal{E}(k, v)$ to denote the set of possible ciphertexts that the value v can be encrypted to under the key k, and $u \in \mathcal{E}(k, v)$ to denote that u is such a ciphertext.
- $\mathcal{D}$ is a deterministic decryption algorithm that takes a key and a ciphertext and returns a bit string—the decryption of the ciphertext—or fails. Moreover, $\mathcal{D}$ is a keyed left inverse of $\mathcal{E}$, i.e., $u \in \mathcal{E}(k, v) \Longrightarrow \mathcal{D}(k, u) = v$, and only ciphertexts can be decrypted, i.e., $\mathcal{D}(k, u) = v \Longrightarrow u \in \mathcal{E}(k, v)$.

Let $\mathcal{S}_{\mathtt{LK}}$ represent the encryption scheme associated with the low key level and similarly let $\mathcal{S}_{\mathtt{HK}}$ represent the high encryption scheme.

The reason for the use of two different encryption schemes for different security levels is to lay the ground for an extension of the system into a general *multi-level* system, i.e., a system with more than two security levels. The standard way of defining security for a multi-level system is in terms of a two-level system. This is done by demanding that for a program to be secure in the multi-level system it should be secure in the two-level system for all order preserving mappings from the multi-level lattice into the two-level lattice.

The idea of the low and high encryption schemes is to represent the mapping of the multi-level system described above. Thus, the low encryption scheme represents the untrusted encryption schemes (for the given mapping) and the high encryption scheme represents the trusted ones. For this reason, we assume different properties of the two encryption schemes. We discuss these properties in detail in Section 3.

To be able to use our encryption schemes on values in general we need a way to encode integers, keys and pairs of values into bit strings and to decode bit strings back into values, when possible. We assume two functions: encode, that takes a value and returns its bit-string representation, and decode, that takes a bit string and returns the represented value, if any. Obviously, $encode$ and $decode$ should be inverses w.r.t. the decodable subdomain of bit strings. In the following, we shall assume that the encryption and decryption functions have been lifted to values by proper composition with the encode and decode functions, so that encryption takes a value and returns a bit string and decryption takes a bit string and returns a value. Thus, $encode$ and $decode$ will not be explicitly mentioned.

2.2.2 Environments

Input and output are modeled in terms of streams of values with the cons operation "$\cdot$" and the distinguished empty stream ϵ. This is a simplified model of input/output, where the input from the environment does not explicitly depend on preceding output to the environment. However, we expect that such a simplification has no impact on the security of isolated actors. Indeed, Clark and Hunt [11] observe that, thanks to the quantification over all streams, it makes no difference whether input/output is modeled in terms of strategies or streams as long as the program is deterministic. We expect this observation to apply to our setting, since the nondeterminism in our language is limited to encryption primitives, and programs may not branch on ciphertexts. Hence, the control flow of the program as well as its resulting non-ciphertext values remain deterministic.

Let ks range over finite streams of keys, and vs over finite streams of values. Finite streams are sufficient, since our semantics only models finitely running programs (cf. Sections 2.3 and 2.4). The *full environment* $E = (M, G, I, O)$ consists of four components: (i) the variable environment M, which is a mapping from variable names to values; (ii) the key-stream environment G, which maps encryption scheme levels to *streams of keys* generated by successive use of the key generator; (iii) the input environment I and (iv) the output environment O, which map channel names to streams of values.

2.3 Semantics of expressions

The evaluation of expressions has the form $\langle M, e\rangle \Downarrow v$: evaluating the expression e in the variable environment M yields the value v. The semantics of integers, variables, total binary operators, pair formation, and projection are entirely standard. Figure 3 presents the semantic rules for expressions.

The rules specific to the treatment of cryptography are encryption (S-ENC) and decryption (S-DEC) which both use the encryption schemes $\mathcal{S}_\gamma$ introduced above. Since $\mathcal{E}$ is nondeterministic, i.e., it returns a set of different ciphertexts for the same key and plaintext, the semantics becomes nondeterministic—there is one separate semantic rule per possible ciphertext.

$$\text{(S-INT)}\frac{}{\langle M, n\rangle \Downarrow n} \qquad \text{(S-VAR)}\frac{M(x) = v}{\langle M, x\rangle \Downarrow v}$$

$$\text{(S-OP)}\frac{\langle M, e_1\rangle \Downarrow v_1 \qquad \langle M, e_2\rangle \Downarrow v_2 \qquad v = v_1\ op\ v_2}{\langle M, e_1\ op\ e_2\rangle \Downarrow v}$$

$$\text{(S-PAIR)}\frac{\langle M, e_1\rangle \Downarrow v_1 \qquad \langle M, e_2\rangle \Downarrow v_2}{\langle M, (e_1, e_2)\rangle \Downarrow (v_1, v_2)}$$

$$\text{(S-FST)}\frac{\langle M, e\rangle \Downarrow (v_1, v_2)}{\langle M, \texttt{fst}(e)\rangle \Downarrow v_1} \qquad \text{(S-SND)}\frac{\langle M, e\rangle \Downarrow (v_1, v_2)}{\langle M, \texttt{snd}(e)\rangle \Downarrow v_2}$$

$$\text{(S-ENC)}\frac{\langle M, e_1\rangle \Downarrow k \qquad \langle M, e_2\rangle \Downarrow v \qquad k \in Key_\gamma \qquad u \in \mathcal{E}_\gamma(k, v)}{\langle M, \texttt{encrypt}_\gamma\ (e_1, e_2)\rangle \Downarrow u}$$

$$\text{(S-DEC)}\frac{\langle M, e_1\rangle \Downarrow k \qquad \langle M, e_2\rangle \Downarrow u \qquad k \in Key_\gamma \qquad v = \mathcal{D}_\gamma(k, u)}{\langle M, \texttt{decrypt}_\gamma\ (e_1, e_2)\rangle \Downarrow v}$$

Fig. 3. Semantics of expressions

2.4 Semantics of commands

Figure 4 presents the semantics for commands. Commands are state transformers of the form $\langle E, c\rangle \Downarrow E'$: the command c yields the new environment E' when run in the environment E. The semantics of the commands is mostly standard for a while language with channels—the only rule specific to encryption is the rule for key generation (S-NEWKEY). It takes a variable and a level of the key to be generated and assigns the topmost element in the key stream associated to that level in the key-stream environment to that variable. Input and output are both provided as commands (S-INPUT and S-OUTPUT).

3 Security

This section introduces possibilistic noninterference as a semantic model of security for programs with encryption. We begin by stating two assumptions we make on encryption and decryption. Thereafter we argue that standard noninterference cannot be used to model encryption, because of a problem known as *occlusion*. Instead, we suggest possibilistic noninterference and show how this can be used together with the assumed properties of the encryption scheme to model safe uses of encryption without introducing occlusion. The section concludes by investigating the relation between our assumptions and common cryptographic attacker models.

$$\text{(S-SKIP)}\frac{}{\langle E, \texttt{skip}\rangle \Downarrow E} \qquad \text{(S-SEQ)}\frac{\langle E, c_1\rangle \Downarrow E' \quad \langle E', c_2\rangle \Downarrow E''}{\langle E, c_1; c_2\rangle \Downarrow E''}$$

$$\text{(S-VARASGN)}\frac{\langle M, e\rangle \Downarrow v}{\langle (M, G, I, O), x := e\rangle \Downarrow (M[x \mapsto v], G, I, O)}$$

$$\text{(S-NEWKEY)}\frac{G(\gamma) = k \cdot ks}{\langle (M, G, I, O), \texttt{newkey}(x, \gamma)\rangle \Downarrow (M[x \mapsto k], G[\gamma \mapsto ks], I, O)}$$

$$\text{(S-IF1)}\frac{\langle M, e\rangle \Downarrow v \quad v \neq 0 \quad \langle (M, G, I, O), c_1\rangle \Downarrow E'}{\langle (M, G, I, O), \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2\rangle \Downarrow E'}$$

$$\text{(S-IF2)}\frac{\langle M, e\rangle \Downarrow 0 \quad \langle (M, G, I, O), c_2\rangle \Downarrow E'}{\langle (M, G, I, O), \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2\rangle \Downarrow E'}$$

$$\text{(S-WHILE1)}\frac{\langle M, e\rangle \Downarrow v \quad v \neq 0 \quad \langle (M, G, I, O), c; \texttt{while } e \texttt{ do } c\rangle \Downarrow E'}{\langle (M, G, I, O), \texttt{while } e \texttt{ do } c\rangle \Downarrow E'}$$

$$\text{(S-WHILE2)}\frac{\langle M, e\rangle \Downarrow 0}{\langle (M, G, I, O), \texttt{while } e \texttt{ do } c\rangle \Downarrow (M, G, I, O)}$$

$$\text{(S-INPUT)}\frac{I(ch) = v \cdot vs}{\langle (M, G, I, O), \texttt{in}(x, ch)\rangle \Downarrow (M[x \mapsto v], G, I[ch \mapsto vs], O)}$$

$$\text{(S-OUTPUT)}\frac{\langle M, e\rangle \Downarrow v}{\langle (M, G, I, O), \texttt{out}(ch, e)\rangle \Downarrow (M, G, I, O[ch \mapsto v \cdot O[ch]])}$$

Fig. 4. Semantics of commands

3.1 Assumptions on the cryptographic primitives

We begin with an informal discussion of two properties of the encryption schemes that we need. We use these properties to formulate and prove possibilistic noninterference for our system. Section 3.5 discusses how these properties relate to standard computational properties.

The first property is a *confidentiality property*. It states the *indistinguishability* of ciphertexts for high encryption schemes. Intuitively, this captures the attacker's capability to learn anything about the plaintext by observing the ciphertext. In particular, indistinguishability gives us full freedom in the process of defining a low-equivalence relation for ciphertexts. We do not assume indistinguishability of the ciphertexts of the low encryption scheme.

The second property is an *authenticity property* needed in the treatment of decryption. More precisely we are assuming that decryption using the *wrong* key fails:

$$u \in \mathcal{E}(k', v) \Longrightarrow \mathcal{D}(k, u) = \bot \text{ if } k \neq k'$$

We assume that the same property holds for the decryption function of the low encryption scheme.

The assumption that encryption with the wrong key fails gives us that each ciphertext is successfully decryptable with one unique key, which is captured by the following lemma:

Lemma 1 *Uniqueness of decryption keys.*

$$\mathcal{D}(k_1, u) = v_1 \wedge \mathcal{D}(k_2, u) = v_2 \wedge v_1 \neq \bot \wedge v_2 \neq \bot \Longrightarrow k_1 = k_2$$

Proof. $\mathcal{D}(k_1, u) = v_1$ gives that $u \in \mathcal{E}(k_1, v_1)$, since only ciphertexts are decryptable. Now, assume that $k_1 \neq k_2$; the property that encryption with the wrong key fails gives $\mathcal{D}(k_2, u) = \bot$ from $u \in \mathcal{E}(k_1, v_1)$. However, this contradicts the assumption that $\mathcal{D}(k_2, u) = v_2$, and $v_2 \neq \bot$. Hence $k_1 = k_2$. $\square$

Note that this lemma implies a uniqueness property of decryption results, i.e., not only can we show $k_1 = k_2$ under the assumptions of the lemma but also $v_1 = v_2$, since decryption is deterministic.

3.2 Insufficiency of standard noninterference

The prevailing notion when defining confidentiality in the analysis of information flows is noninterference. Noninterference is typically formalized as the preservation of a *low-equivalence* relation under the execution of a program: if a program is run in two low-equivalent environments then the resulting environments should be low-equivalent. For ordinary values, like integers, low-equivalence demands that low values are equal. From the assumption that ciphertexts are indistinguishable we are free to chose any low-equivalence relation for ciphertexts. For instance, one may consider it secure to treat all ciphertexts as low-equivalent. However appealing this may be, such a treatment leads to the ability of masking implicit flows in ciphertexts. Consider the program in Listing 1 for some high encryption key stored in the variable k and some value stored in the variable a.

```
l₁:= encrypt_HK(k, a);
if h then l₂ := encrypt_HK(k, a)
      else l₂ := l₁;
```

Listing 1. Occlusion

If all encrypted values are considered low-equivalent then we cannot distinguish between the two low variables l_1 and l_2 even though it is clear that the equality/inequality of the first and the second value reflects the secret value h. This is an instance of a general problem known as *occlusion* [32] (which we recall from the introduction), where masking an intended information flow in an indistinguishability definition may also mask other unintended leaks. In the rest of the article, we will refer to the problem illustrated in Listing 1 as the *occlusion problem* or, simply, occlusion.

For standard noninterference, any other relation than the one that simply relates all ciphertexts removes the possibility to consider secure uses of encryption for noninterference. Instead, we use a variant of noninterference known as possibilistic noninterference, which allows us to create a notion of low-equivalence that semantically rejects occlusion without preventing intuitively secure uses.

3.3 Possibilistic noninterference

Let Σ denote the type environment under which programs are run and let $E_1 \sim_\Sigma E_2$ denote that the environments E_1 and E_2 are low-equivalent w.r.t the type environment Σ. Section 4 defines how type environments are built. For now we only use that if $E_1 \sim_\Sigma E_2$ then environments E_1 and E_2 are low-equivalent, i.e., indistinguishable for an attacker. A pair of commands c_1 and c_2 are possibilistically noninterfering if

$$\begin{aligned} NI(c_1, c_2)_\Sigma \equiv\ & \forall E_1, E_2 \mathrel{.} E_1 \sim_\Sigma E_2 \wedge \\ & \langle E_1, c_1\rangle \Downarrow \hat{E}_1 \wedge \hat{E}_1 \neq \emptyset \wedge \langle E_2, c_2\rangle \Downarrow \hat{E}_2 \wedge \hat{E}_2 \neq \emptyset \implies \\ & \forall E_1' \in \hat{E}_1 \mathrel{.} \exists E_2' \in \hat{E}_2 \mathrel{.} E_1' \sim_\Sigma E_2' \wedge \forall E_2' \in \hat{E}_2 \mathrel{.} \exists E_1' \in \hat{E}_1 \mathrel{.} E_1' \sim_\Sigma E_2' \end{aligned}$$

where the evaluation relation is lifted to a set of results as follows:

$$\langle E, c\rangle \Downarrow \hat{E} \text{ iff } \hat{E} = \{E' \mid \langle E, c\rangle \Downarrow E'\}$$

Intuitively, for every pair of low-equivalent environments in which the commands terminate it holds that there exists a *possibility* that each environment produced by the first command when run in the first environment can be produced by the second command when run in the second environment.

A program c is considered *secure* w.r.t. Σ if it is noninterfering with itself, i.e., $NI(c, c)_\Sigma$.

It is straightforward to see that the noninterference relation is symmetric and transitive (because the low-equivalence relation $\sim_\Sigma$ is symmetric and transitive), but not reflexive. If it were reflexive then all programs would be considered secure.

By only considering environments for which the commands terminate, we ignore crash-related leaks.

3.4 *Adequacy of possibilistic noninterference*

The choice of possibilistic noninterference does not automatically solve the above problem—like before, using the full low-equivalence relation on ciphertexts leads to occlusion. Let $\doteq$ denote the low-equivalence relation for ciphertexts, used in the definition of low-equivalence for environments $\sim_\Sigma$ (via low-equivalence for values $\sim_\tau$; see Section 4.2 below). To allow for secure usages of encryption, while at the same time protecting from occlusion we put the following demands on $\doteq$:

$$u_1 \in \mathcal{E}_{\mathtt{HK}}(k_1, v_1) \Longrightarrow \exists u_2 \,.\, u_2 \in \mathcal{E}_{\mathtt{HK}}(k_2, v_2) \wedge u_1 \doteq u_2 \tag{1}$$

$$\exists u_1, u_2 \,.\, u_1 \in \mathcal{E}_{\mathtt{HK}}(k_1, v_1) \wedge u_2 \in \mathcal{E}_{\mathtt{HK}}(k_2, v_2) \wedge u_1 \not\doteq u_2 \tag{2}$$

The first demand ensures the possibility for safe usages, while the second one excludes occlusion. In addition to this we demand that $\doteq$ is an equivalence relation, and that $\doteq$ does not relate ciphertexts of different sizes. The former of the additional demands allows the low-equivalence relation $\sim_\Sigma$ to be an equivalence relation, and the latter is needed since we do not assume that the cryptographic primitives hide the length of the plaintexts. If such an assumption is made (as it is the case in Laud's work [22]), the latter demand can be safely dropped.

3.4.1 *Safe uses of encryption*

In the setting of possibilistic noninterference we must make sure that any ciphertext produced by each plaintext and key has a low-equivalent ciphertext for any other choice of plaintext and key. To see this consider the program $l := \mathit{encrypt}_{\mathtt{HK}}(k, a)$ for some high key k and some secret value a. Assume that E_1 and E_2 are two low-equivalent environments such that $E_1(a) = v_1$, $E_2(a) = v_2$, $E_1(k) = k_1$, and $E_2(k) = k_2$. The execution of $l := \mathit{encrypt}_{\mathtt{HK}}(k, a)$ yields:

$$\hat{E}_1 = \{E_1[l \mapsto u] \mid u \in \mathcal{E}_{\mathtt{HK}}(k_1, v_1)\}, \text{ and } \hat{E}_2 = \{E_2[l \mapsto u] \mid u \in \mathcal{E}_{\mathtt{HK}}(k_2, v_2)\}$$

Clearly, given the property expressed by Equation 1 for every environment in $\hat{E}_1$ there is a low-equivalent environment in $\hat{E}_2$.

3.4.2 *Exclusion of occlusion*

To see that Equation 2 excludes occlusion, let us consider the execution of the occlusion example in Listing 1. To argue by contradiction, suppose the program is noninterfering. Assume now that E_1 and E_2 are two low-equivalent environments such that $E_1(h) = \mathit{false}$, $E_2(h) = \mathit{true}$, $E_1(a) = v_1$, $E_2(a) = v_2$, $E_1(k) = k_1$, and $E_2(k) = k_2$. The result of running the program in E_1 is the set

$$\hat{E}_1 = \{E_1[l_1 \mapsto u, l_2 \mapsto u] \mid u \in \mathcal{E}_{\mathtt{HK}}(k_1, v_1)\}$$

and the result of running it in the second environment E_2 is the set

$$\hat{E}_2 = \{E_2[l_1 \mapsto u_1, l_2 \mapsto u_2] \mid u_1 \in \mathcal{E}_{\texttt{HK}}(k_2, v_2), u_2 \in \mathcal{E}_{\texttt{HK}}(k_2, v_2)\}$$

Possibilistic noninterference demands that for each environment in $\hat{E}_2$ there exists a low-equivalent environment in $\hat{E}_1$. Let us pick an environment from $\hat{E}_2$ where $u_1 \in \mathcal{E}_{\texttt{HK}}(k_2, v_2)$ and $u_2 \in \mathcal{E}_{\texttt{HK}}(k_2, v_2)$ such that $u_1 \not\doteq u_2$. Such u_1 and u_2 exist by Equation 2. By possibilistic noninterference, there exists an environment E_1' in $\hat{E}_1$ such that $E_1'(l_1) \doteq u_1$ and $E_1'(l_2) \doteq u_2$. However, by construction of $\hat{E}_1$ we have that $E_1'(l_1) = E_1'(l_2) = u$ for some u. We have $u \doteq u_1$ and $u \doteq u_2$. This implies $u_1 \doteq u_2$ by transitivity, which contradicts the initial choice of u_1 and u_2.

3.4.3 Plausibility of low-equivalence properties

Now, it remains to show that the properties of Equations 1 and 2 are plausible for ciphertexts originating from existing cryptographic primitives. We do this by showing that for probabilistic symmetric encryption schemes we can easily form a low-equivalence relation satisfying Equations 1 and 2.

Probabilistic symmetric encryption schemes generate a random initial vector iv on every invocation of the encryption function $\mathsf{ENC}_{\mathit{Impl}}$, where the value of iv is used in computing the ciphertext for plaintext v using key k, and $\mathcal{E}(k, v) = \cup_{iv}\mathsf{ENC}_{\mathit{Impl}}(iv, k, v)$. We define low-equivalence $\doteq$ by relating equally-sized ciphertexts *with the same random initial vector*:

$$\forall k_1, k_2, v_1, v_2 \; . \; \mathsf{ENC}_{\mathit{Impl}}(iv, k_1, v_1) \doteq \mathsf{ENC}_{\mathit{Impl}}(iv, k_2, v_2)$$

Since this relation relates two ciphertexts if and only if they were created with the same initial vector, we have that for any choice of plaintext and key there will be exactly one related ciphertext for any other plaintext and key. The existence guarantees Equation 1, and the uniqueness guarantees Equation 2, since there is more than one possible initial vector.

To exemplify this, consider cipher-block chaining (CBC) with a random initial vector encryption scheme [29]. The encryption algorithm in this scheme is as a two-step process: (i) generating a random initial vector iv, and (ii) using the generated iv to compute the ciphertext. Let $\mathsf{CBC}_{\mathit{Impl}}(iv, k, v)$ correspond to the second step of the implementation; the ciphertext returned by the encryption algorithm is a tuple of the form (iv, u), where iv is generated in the first step and $u = \mathsf{CBC}_{\mathit{Impl}}(iv, k, v)$ is computed in the second one. Then we can define two ciphertexts (iv_1, u_1) and (iv_2, u_2) produced by this scheme as low-equivalent ($\doteq$) if they are equally-sized and agree upon initial vectors, i.e., $iv_1 = iv_2$. Putting this definition to the test with the occlusion example, we can see that the occlusion example is indeed rejected for the same general reasons as in the previous subsection.

Recently, Laud [22] has investigated under which conditions our possibilistic noninterference of Section 3.3 implies computational security. It is reassuring that Laud's requirements on the underlying encryption scheme are essentially those that we conjectured in an earlier version [2] of this article: (i) indistinguishability of ciphertexts under chosen plaintext attacks (IND-CPA), which provides ciphertexts confidentiality, and (ii) integrity of plaintexts (INT-PTXT), which achieves authenticity. In Laud's work, these notions are strengthened compared to their standard definitions to also hide key identities [7]. Laud points out that such primitives can be easily constructed in the *random oracle models* [6].

While the language of [22] is simpler compared to the one presented here, we believe his proof technique straightforwardly generalizes to our full system. With such a result at hand, we are ready to capitalize on the modularity of our approach. For a given language and type system, as soon as we can prove that all well-typed programs are noninterfering, we automatically get computational security. This opens up possibilities for reasoning about expressive languages and type systems, where all we have to worry about are noninterference proofs (which are typically simpler than proofs of computational soundness).

4 Types

The syntax of the types is defined in Figure 2. A *primitive type* is either a *security annotated basic type*, a pair of primitive types or a *key type*. The security annotation assigns a security level to the basic type expressing whether it is *high* or *low*. The types of encrypted values are *structural* in the sense that the type reflects the original type of the encrypted values as well as the level of the key that was used in the encryption. For instance, $\texttt{enc}_{\texttt{HK}}\ (\texttt{int H})\ \texttt{L}$ is the type of a high integer that has been encrypted with a high key once and $\texttt{enc}_{\texttt{HK}}\ (\texttt{enc}_{\texttt{HK}}\ (\texttt{int H})\ \texttt{L})\ \texttt{L}$ is the type of an integer that has been encrypted with a high key twice. The type of the variable environment Ω is a map from variables to primitive types, the type of the input environment and the output environment Θ is a map from channel names to primitive types, and the key-stream environment defines its own type (in the domain of the environment). The type of the entire environment, $\Sigma = (\Omega, \Theta)$, is the pair of a variable type environment and a channel type environment.

$$\text{(WF-INT)}\frac{}{n : \mathtt{int}\ \sigma} \qquad \text{(WF-PAIR)}\frac{v_1 : \tau_1 \quad v_2 : \tau_2}{(v_1, v_2) : (\tau_1, \tau_2)}$$

$$\text{(WF-KEY-L)}\frac{k \in Key_{\mathtt{LK}}}{k : \mathtt{key\ LK}} \qquad \text{(WF-KEY-H)}\frac{k \in Key_{\mathtt{HK}}}{k : \mathtt{key\ HK}}$$

$$\text{(WF-ENC)}\frac{k : \mathtt{key}\ \gamma \quad u \in \mathcal{E}_\gamma(k, v) \quad v : \tau}{u : \mathtt{enc}_\gamma\ \tau\ \sigma}$$

$$\text{(WF-MEM)}\frac{\forall x \in dom(\Omega)\ \ M(x) : \Omega(x)}{M : \Omega}$$

$$\text{(WF-IOENV)}\frac{\forall ch \in dom(\Theta)\ .\ X(ch) : \Theta(ch)}{X : \Theta}, \quad X \in \{I, O\}$$

$$\text{(WF-KEYGEN)}\frac{\forall \gamma \in \{\mathtt{LK}, \mathtt{HK}\}\ .\ G(\gamma) : \mathtt{key}\ \gamma}{G}$$

$$\text{(WF-STR1)}\frac{}{\epsilon : \tau} \qquad \text{(WF-STR2)}\frac{v : \tau \quad vs : \tau}{v \cdot vs : \tau}$$

$$\text{(WF-ENV)}\frac{M : \Omega \quad G \quad I : \Sigma \quad O : \Sigma}{(M, G, I, O) : (\Omega, \Sigma)}$$

Fig. 5. Well-formedness

4.1 Well-formed values

Well-formedness defines the meaning of the non-security part of the types, by telling what values have what types. The well-formedness relation is defined in Figure 5.

The integers are the only well-formed values of type int. The set of low keys and the set of high keys form the well-formed values of the low-key type and the high-key type respectively. An encrypted value is well-formed w.r.t. an encryption type if there exists a well-formed key that decrypts the encrypted value to a well-formed value. A pair is well-formed w.r.t. a pair type if both parts of the pair are well-formed w.r.t. the corresponding part of the pair type. Streams, i.e., the input environment, the output environment and the key streams, are well-formed if all their elements are well-formed in the type corresponding to the stream. Finally, the environments are well-formed if all their parts are well-formed.

4.2 Low-equivalence

Figure 6 contains the low-equivalence relation. For complex types, i.e., pairs, environments and streams, low-equivalence is defined structurally by demanding the parts of the complex type to be low-equivalent w.r.t. the corresponding types. Any

$$\text{(LE-KEY-L)}\frac{}{k_{\mathtt{LK}} \sim_{\mathtt{key\ LK}} k_{\mathtt{LK}}} \qquad \text{(LE-KEY-H)}\frac{}{k_{\mathtt{HK}1} \sim_{\mathtt{key\ HK}} k_{\mathtt{HK}2}}$$

$$\text{(LE-INT-L)}\frac{}{n \sim_{\mathtt{int\ L}} n} \qquad \text{(LE-INT-H)}\frac{}{n_1 \sim_{\mathtt{int\ H}} n_2}$$

$$\text{(LE-ENC-H)}\frac{}{u_1 \sim_{\mathtt{enc}_\gamma\ \tau\ \mathtt{H}} u_2} \qquad \text{(LE-PAIR)}\frac{v_{11} \sim_{\tau_1} v_{21} \qquad v_{12} \sim_{\tau_2} v_{22}}{(v_{11}, v_{12}) \sim_{(\tau_1,\tau_2)} (v_{21}, v_{22})}$$

$$\text{(LE-MEM)}\frac{\forall x \in \mathtt{dom}\ (\Omega) \qquad M_1(x) \sim_{\Omega(x)} M_2(x)}{M_1 \sim_\Omega M_2}$$

$$\text{(LE-IOENV)}\frac{\forall ch \in dom(\Theta)\ .\ X_1(ch) \sim_{\Theta(ch)} X_2(ch)}{X_1 \sim_\Theta X_2}, (X_1,X_2) \in \{(I_1,I_2),(O_1,O_2)\}$$

$$\text{(LE-KGEN)}\frac{\forall \gamma \in \{\mathtt{LK}, \mathtt{HK}\}\ .\ G_1(\gamma) \sim_{\mathtt{key}\ \gamma} G_2(\gamma)}{G_1 \sim G_2}$$

$$\text{(LE-STR1)}\frac{}{\epsilon \sim_\tau \epsilon} \qquad \text{(LE-STR2)}\frac{v_1 \sim_\tau v_2 \qquad vs_1 \sim_\tau vs_2}{v_1 \cdot vs_1 \sim_\tau v_2 \cdot vs_2}$$

$$\text{(LE-ENC-L1)}\frac{\begin{array}{c}\exists v_i, k_i\ .\ v_i = \mathcal{D}_{\mathtt{HK}}(k_i, u_i) \qquad i = 1,2 \\ k_1 \sim_{\mathtt{key\ HK}} k_2 \qquad v_1 \sim_\tau v_2 \qquad u_1 \doteq u_2\end{array}}{u_1 \sim_{\mathtt{enc}_{\mathtt{HK}}\ \tau\ \mathtt{L}} u_2}$$

$$\text{(LE-ENC-L2)}\frac{\begin{array}{c}\exists v_i, k_i\ .\ v_i = \mathcal{D}_{\mathtt{LK}}(k_i, u_i) \qquad i = 1,2 \\ k_1 \sim_{\mathtt{key\ LK}} k_2 \qquad v_1 \sim_{tolow(\tau)} v_2\end{array}}{u_1 \sim_{\mathtt{enc}_{\mathtt{LK}}\ \tau\ \mathtt{L}} u_2}$$

Fig. 6. Low-equivalence

values are low-equivalent w.r.t. a high type. Integers are low-equivalent w.r.t. a low-integer type if they are equal. Low-equivalence for keys is slightly different since keys are not annotated with a security level—only a key level—whose meaning is defined by well-formed values as different sets. Even though it is semantically meaningful to add a security level to key types—the values of keys can be indirectly affected by computation—we have chosen not to. Instead, a low key is considered to be of low security and a high key of high security. Thus, low keys are low-equivalent if they are equal, and any two high keys are low-equivalent.

The most interesting rules are the rules defining low-equivalence w.r.t. a low-encryption type (LE-ENC-L1) and (LE-ENC-L2). Common to both rules is that there must exist a pair of low-equivalent keys w.r.t. the key type of the encryption type that successfully decrypt the encrypted values. The differences between the rules reflect the difference between encryption with a high and a low key.

Rule (LE-ENC-L1) for encryption with a high key connects the low-equivalence relation $\doteq$ on ciphertexts with low-equivalence on values $\sim_\tau$. In (LE-ENC-L1), the encrypted values must be low-equivalent w.r.t. the low-equivalence relation $\doteq$, and

$$\text{(SUB-INT)}\frac{\sigma_1 \sqsubseteq \sigma_2}{\texttt{int}\ \sigma_1 <: \texttt{int}\ \sigma_2} \qquad \text{(SUB-ENC)}\frac{\tau_1 <: \tau_2 \qquad \sigma_1 \sqsubseteq \sigma_2}{\texttt{enc}_\gamma\ \tau_1\ \sigma_1 <: \texttt{enc}_\gamma\ \tau_2\ \sigma_2}$$

$$\text{(SUB-PAIR)}\frac{\tau_1 <: \tau_3 \qquad \tau_2 <: \tau_4}{(\tau_1, \tau_2)\ <: (\tau_3, \tau_4)} \qquad \text{(SUB-KEY)}\frac{}{\texttt{key}\ \gamma <: \texttt{key}\ \gamma}$$

Fig. 7. Subtyping

the values to be encrypted should be low-equivalent w.r.t. the primitive type, τ, of the encryption type.

Since ciphertexts created by low keys can be decrypted by anyone (we assume that the low keys are freely available), we have to demand that the inside of the encrypted value contains only low values. This is done in the (LE-ENC-L2) rule, which demands that the inside is not only low-equivalent w.r.t. its type τ, but low-equivalent w.r.t. $tolow(\tau)$, which is defined as follows:

$$tolow(t\ \sigma) = t\ \texttt{L} \qquad tolow(\texttt{key LK}) = \texttt{key LK}$$
$$tolow((\tau_1, \tau_2)) = (tolow(\tau_1), tolow(\tau_2))$$

Observe that $tolow(\texttt{key HK})$ is undefined to exclude the possibility of encrypting a high key with a low one. This way, we make certain that the result of decrypting low-equivalent encrypted values will result in low-equivalent values and that high values are not stored inside encrypted values that are created by low keys.

4.3 Subtyping

The subtyping is entirely standard; it allows low information to be seen as high with the exception of invariant subtyping for keys. The subtyping relation for primitive types, $<:$, and the subtyping relation for security levels, $\sqsubseteq$, defines the corresponding join operators. The rules for subtyping can be found in Figure 7.

4.4 Expression type rules

The type rules for expressions, defined in Figure 8, are of the form $\Omega \vdash e : \tau$. Encryption with high keys will always result in low encrypted values. Encryption with low keys is possible on any value but produces a result that is as secret as the original value. The type rule for low encryption makes use of function $lvl(\cdot)$ that

$$\text{(T-INT)}\frac{}{\Omega \vdash n : \texttt{int L}} \qquad \text{(T-VAR)}\frac{\Omega(x) = \tau}{\Omega \vdash x : \tau}$$

$$\text{(T-PAIR)}\frac{\Omega \vdash e_1 : \tau_1 \qquad \Omega \vdash e_2 : \tau_2}{\Omega \vdash (e_1, e_2) : (\tau_1, \tau_2)}$$

$$\text{(T-FST)}\frac{\Omega \vdash e : (\tau_1, \tau_2)}{\Omega \vdash \texttt{fst}(e) : \tau_1} \qquad \text{(T-SND)}\frac{\Omega \vdash e : (\tau_1, \tau_2)}{\Omega \vdash \texttt{snd}(e) : \tau_2}$$

$$\text{(T-OP)}\frac{\Omega \vdash e_1 : \texttt{int}\ \sigma_1 \qquad \Omega \vdash e_2 : \texttt{int}\ \sigma_2}{\Omega \vdash e_1\ \mathit{op}\ e_2 : \texttt{int}\ \sigma_1 \sqcup \sigma_2}$$

$$\text{(T-ENC1)}\frac{\Omega \vdash e_1 : \texttt{key HK} \qquad \Omega \vdash e_2 : \tau}{\Omega \vdash \texttt{encrypt}_{\texttt{HK}}\ (e_1, e_2) : \texttt{enc}_{\texttt{HK}}\ \tau\ \texttt{L}}$$

$$\text{(T-ENC2)}\frac{\Omega \vdash e_1 : \texttt{key LK} \qquad \Omega \vdash e_2 : \tau \qquad \mathit{lvl}(\tau) = \sigma}{\Omega \vdash \texttt{encrypt}_{\texttt{LK}}\ (e_1, e_2) : \texttt{enc}_{\texttt{LK}}\ \tau\ \sigma}$$

$$\text{(T-DEC)}\frac{\Omega \vdash e_1 : \texttt{key}\ \gamma \qquad \Omega \vdash e_2 : \texttt{enc}_{\gamma}\ \tau\ \sigma}{\Omega \vdash \texttt{decrypt}_{\gamma}\ (e_1, e_2) : \tau^{\sigma}}$$

Fig. 8. Type rules of expressions

computes the security level of the given value:

$$\mathit{lvl}(t\ \sigma) = \sigma \quad \mathit{lvl}((\tau_1, \tau_2)) = \mathit{lvl}(\tau_1) \sqcup \mathit{lvl}(\tau_2) \quad \mathit{lvl}(\texttt{key LK}) = \texttt{L} \quad \mathit{lvl}(\texttt{key HK}) = \texttt{H}$$

Decryption is allowed only if the key level of the key used for decryption matches the key level of the encrypted value. The result of the decryption is tainted by the security level of the encrypted values. The taint function is defined as follows:

$$(t\ \sigma)^{\sigma'} = t\ (\sigma \sqcup \sigma') \qquad (\tau_1, \tau_2)^{\sigma} = (\tau_1^{\sigma}, \tau_2^{\sigma})$$
$$(\texttt{key LK})^{\texttt{L}} = \texttt{key LK} \qquad (\texttt{key HK})^{\sigma} = \texttt{key HK}$$

Note that the taint function is not defined for $(\texttt{key LK})^{\texttt{H}}$—neither keeping the key level as LK nor raising it to HK is adequate in this case. The reason for this is the choice to not decorate key types with an additional security level as discussed in Section 4.2.

4.5 Command type rules

The type rules for commands, defined in Figure 9, are of the form $\Sigma, pc \vdash c$. As with expressions most of the rules are standard for a security type system (cf. [38]). In order to prevent implicit flows, we adopt the notion of a *security context* [14]. The security context of a program point is defined to be the least upper bound of the security levels of the conditional expressions of the enclosing conditionals. The context affects the commands with side-effects, i.e., variable assignment, key

$$\text{(T-ASGN)}\frac{\Omega \vdash e : \tau \qquad \tau <: \Omega(x) \qquad pc \sqsubseteq lvl(\Omega(x))}{(\Omega, \Theta), pc \vdash x := e}$$

$$\text{(T-NEWKEY)}\frac{pc \sqsubseteq lvl(\texttt{key}\ \gamma) \qquad \Omega(x) = \texttt{key}\ \gamma}{(\Omega, \Theta), pc \vdash \texttt{newkey}(x, \gamma)}$$

$$\text{(T-SKIP)}\frac{}{\Sigma, pc \vdash \texttt{skip}} \qquad \text{(T-SEQ)}\frac{\Sigma, pc \vdash c_1 \qquad \Sigma, pc \vdash c_2}{\Sigma, pc \vdash c_1; c_2}$$

$$\text{(T-OUTPUT)}\frac{\Omega \vdash e : \tau \qquad \tau <: \Theta(ch) \qquad pc \sqsubseteq lvl(\Theta(ch))}{(\Omega, \Theta), pc \vdash \texttt{out}(ch, x)}$$

$$\text{(T-IF)}\frac{\Omega \vdash e : \texttt{int}\ \sigma \qquad (\Omega, \Theta), pc \sqcup \sigma \vdash c_i \;\; i = 1, 2}{(\Omega, \Theta), pc \vdash \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2}$$

$$\text{(T-WHILE)}\frac{\Omega \vdash e : \texttt{int}\ \sigma \qquad (\Omega, \Theta), pc \sqcup \sigma \vdash b}{(\Omega, \Theta), pc \vdash \texttt{while}\ e\ \texttt{do}\ b}$$

$$\text{(T-INPUT)}\frac{pc \sqsubseteq lvl(\Omega(x)) \qquad \Theta(ch) <: \Omega(x)}{(\Omega, \Theta), pc \vdash \texttt{in}(x, ch)}$$

Fig. 9. Type rules of commands

generation, input, and output. The type rule for sequences of statements (T-SEQ) checks both statements of the sequence. *If* and *while* are the two constructs that can lead to implicit flows since they affect the control flow. Thus, the body of the *if* and *while* are checked in the context of the security level of the control expression. This way, when a branch depends on a secret, the body of that branch is prevented from causing any low side effects. The generation of a new key with the requested security level results in a key with that security level if the requested level is not below the context type. The reason for this is that we assume that the low-key stream is publicly observable.

Returning to the occlusion problem, we note that the program in Listing 1 is rejected by our type system. In particular, it is rejected by rule (T-IF) for the same reasons assigning to a low integer would be rejected.

On a general note, it is important to recall that it is not the type rule (T-IF) itself that is central to our contribution, but it is the semantic justification of rules (T-IF) and (T-ENC1) in combination that rules out weaker, occlusion-prone, versions of (T-IF).

5 Soundness

This section introduces and proves a number of relevant properties of the system. In particular, we introduce noninterference for expressions and prove that well-typed expressions are noninterfering, i.e., secure.

We focus on the soundness of the expressions, since all the encryption relevant additions are in the expression language. The security conditions for commands are similar to the ones for expressions; their proofs do not differ from any standard proofs of possibilistic noninterference. The proof for the expressions has been formalized in the proof assistant Coq and is available from the second author's homepage[1]. Below, we detail the parts of the proof specific for the treatment of encryption and decryption; we refer the reader to the formal proof for the details of the standard parts of the proofs.

5.1 Set-lifted semantics

The proof is carried out in a set-lifted version of the semantics in this article for proof-technical reasons; since the standard semantics results in a set of values (because of the nondeterministic model of encryption) it is convenient to work in a set-lifted semantics when using the induction hypotheses.

Thus, in the following we will use an evaluation relation for expressions that relates sets of variable environments to sets of values $\langle \hat{M}, e\rangle \Downarrow \hat{v}$, and, similarly, for statements a relation that relates sets of environments to sets of environments $\langle \hat{E}_1, c\rangle \Downarrow \hat{E}_2$. Figure 10 contains the set-lifted semantics for the expressions, given to support the proofs of this section.

The connection between the original semantics in the article and the set-lifted semantics is captured in the following two lemmas. The first lemma says that the set-lifted semantics models all of the original semantics.

Lemma 2 *Coverage of the proof semantics of expressions.*

$$\langle M, e\rangle \Downarrow v \Rightarrow \exists \widehat{v} \,.\, v \in \widehat{v} \wedge \langle \{M\}, e\rangle \Downarrow \widehat{v}$$

Proof. By induction on the derivation of $\langle M, e\rangle \Downarrow v$. □

The second lemma says that the set semantic does not introduce anything not modeled by the original semantics.

Lemma 3 *Precision of the proof semantics of expressions.*

$$\langle \{M\}, e\rangle \Downarrow \widehat{v} \Longrightarrow \forall v \,.\, v \in \widehat{v} \Longrightarrow \langle M, e\rangle \Downarrow v$$

Proof. By induction on the structure of e. □

[1] http://www.cs.chalmers.se/~utter

$$\text{(SET-INT)}\frac{}{\langle \hat{M}, n\rangle \Downarrow \{n\}} \qquad \text{(SET-VAR)}\frac{\hat{v} = \{v \mid M \in \hat{M},\ M(x) = v\}}{\langle \hat{M}, x\rangle \Downarrow \hat{v}}$$

$$\text{(SET-OP)}\frac{\langle \hat{M}, e_1\rangle \Downarrow \hat{v}_1 \quad \langle \hat{M}, e_2\rangle \Downarrow \hat{v}_2 \quad \hat{v} = \{v_1\ op\ v_2 \mid v_1 \in \hat{v}_1,\ v_2 \in \hat{v}_2\}}{\langle \hat{M}, e_1\ op\ e_2\rangle \Downarrow \hat{v}}$$

$$\text{(SET-PAIR)}\frac{\langle \hat{M}, e_1\rangle \Downarrow \hat{v}_1 \quad \langle \hat{M}, e_2\rangle \Downarrow \hat{v}_2 \quad \hat{v} = \{(v_1, v_2) \mid v_1 \in \hat{v}_1,\ v_2 \in \hat{v}_2\}}{\langle \hat{M}, (e_1, e_2)\rangle \Downarrow \hat{v}}$$

$$\text{(SET-FST)}\frac{\langle \hat{M}, e\rangle \Downarrow \hat{v} \quad \hat{v}_1 = \{v_1 \mid (v_1, v_2) \in \hat{v}\}}{\langle \hat{M}, \texttt{fst}(e)\rangle \Downarrow \hat{v}_1} \qquad \text{(SET-SND)}\frac{\langle \hat{M}, e\rangle \Downarrow \hat{v} \quad \hat{v}_2 = \{v_2 \mid (v_1, v_2) \in \hat{v}\}}{\langle \hat{M}, \texttt{snd}(e)\rangle \Downarrow \hat{v}_2}$$

$$\text{(SET-ENC)}\frac{\langle \hat{M}, e_1\rangle \Downarrow \hat{k} \quad \langle \hat{M}, e_2\rangle \Downarrow \hat{v} \quad \hat{k} \subseteq Key_\gamma \quad \hat{u} = \{u \mid k \in \hat{k},\ v \in \hat{v},\ u \in \mathcal{E}_\gamma(k, v)\}}{\langle \hat{M}, \texttt{encrypt}_\gamma\ (e_1, e_2)\rangle \Downarrow \hat{u}}$$

$$\text{(SET-DEC)}\frac{\langle \hat{M}, e_1\rangle \Downarrow \hat{k} \quad \langle \hat{M}, e_2\rangle \Downarrow \hat{u} \quad k \subseteq Key_\gamma \quad \hat{v} = \{\mathcal{D}_\gamma(k, u) \mid k \in \hat{k},\ u \in \hat{u}\}}{\langle \hat{M}, \texttt{decrypt}_\gamma\ (e_1, e_2)\rangle \Downarrow \hat{v}}$$

Fig. 10. Set-lifted semantics for expressions

5.2 *Leaf-determinism*

We begin by proving an important property of expressions that intuitively is a stronger version of the preservation of types; preservation of types expresses that well-formedness is preserved under execution of well-typed programs.

We define *leaf determinism* as follows

$$\frac{}{\{n\} :: \texttt{int}\ \sigma} \qquad \frac{}{\{k_\gamma\} :: \texttt{key}\ \gamma} \qquad \frac{\{v \mid u \in \mathcal{E}_\gamma(k_\gamma, v), u \in \widehat{u}\} :: \tau}{\widehat{u} :: \texttt{enc}_\gamma\ \tau\ \sigma}$$

$$\frac{\widehat{u} = \{(v_1, v_2) \mid v_1 \in \widehat{u}_1, v_2 \in \widehat{u}_2\} \quad \widehat{u}_1 :: \tau_1 \quad \widehat{u}_2 :: \tau_2}{\widehat{u}_2 :: (\tau_1, \tau_2)}$$

$$\frac{\{M(x) \mid M \in \hat{M}\} :: \Omega(x)}{\hat{M} :: \Omega}$$

This relation is equivalent to a set-lifted version of the standard well-formedness relation with the addition of the demand that any well-formed set of values decrypt to one unique value. The idea is to capture that all nondeterminism comes from the encryption and not from any other source, i.e., that all values apart from encrypted values are deterministic. If $V :: T$ for some set V and some type T we say that V is *leaf deterministic*. To exemplify, if we have a set of keys $\hat{k}$, that is leaf-deterministic, i.e., $\hat{k}$ :: `key` γ we know that it is a singleton set. Furthermore, if we have a set of encrypted values $\hat{u}$ that is leaf-deterministic w.r.t. $\texttt{enc}_\gamma$ `int` σ, say, $\widehat{u}$ :: $\texttt{enc}_\gamma$ `int` σ that set is not a singleton, but we know that all values in the set decrypt to the same integer, since the set $\{v \mid u \in \mathcal{E}_\gamma(k, v), u \in \widehat{u}\}$ used to form $\hat{u}$ is leaf-deterministic w.r.t `int`, which means it is a singleton set.

The connection between leaf determinism and well-formedness is captured by the following two lemmas. First, as hinted above, leaf determinism implies well-formedness.

Lemma 4 *Leaf determinism implies well-formedness, i.e.,* $\widehat{v} :: \tau \Longrightarrow \forall v \in \widehat{v} \,.\, v : \tau$

Proof. By induction on the derivation of $\widehat{v} :: \tau$. □

The connection holds the other way around as well. Given a well-formed value, the singleton set of that value is leaf deterministic, i.e.:

Lemma 5 *Well-formed singletons are leaf deterministic, i.e.,* $v : \tau \Longrightarrow \{v\} :: \tau$.

Proof. By induction on the derivation of $v : \tau$. □

With this we can prove that expressions preserve leaf determinism, i.e., that if a well-typed expression e is evaluated in a leaf deterministic environment the result is leaf deterministic.

Theorem 6 *Preservation of leaf determinism of expressions.*

$$\Omega \vdash e : \tau \wedge \hat{M} :: \Omega \wedge \langle \hat{M}, e\rangle \Downarrow \hat{v} \wedge \hat{v} \neq \emptyset \Longrightarrow \hat{v} :: \tau$$

Proof. By induction on the derivation of $\Omega \vdash e : \tau$; let the assumptions be labeled as follows: (2) $\Omega \vdash e : \tau$, (3) $\hat{M} :: \Omega$, (4) $\langle \hat{M}, e\rangle \Downarrow \hat{v}$, and (5) $\hat{v} \neq \emptyset$. We want to show that $\hat{v} :: \tau$. We only give the cases relating to encryption, i.e. (T-ENC1), (T-ENC2), and (T-DEC). We refer the reader to the formal proof for the proofs for (T-INT), (T-VAR), (T-PAIR), (T-FST), (T-SND), and (T-OP). Let $i \in \{1, 2\}$ in the following.

(T-ENC1) We have that $e = \texttt{encrypt}_{\texttt{HK}}\ (e_1, e_2)$, (6) $\Omega \vdash e_1 : \texttt{key HK}$, and (7) $\Omega \vdash e_2 : \tau$. Now, (4) gives (8) $\langle \hat{M}, e_1\rangle \Downarrow \hat{k}$, (9) $\langle \hat{M}, e_2\rangle \Downarrow \hat{v}'$, (10) $\hat{k} \subseteq Key_{\texttt{HK}}$, and (S1) $\hat{v} = \{v \mid k \in \hat{k},\ v' \in \hat{v}',\ v \in \mathcal{E}_\gamma(k, v')\}$.

We want to show that $\hat{v} :: \texttt{enc}_{\texttt{HK}}\ \tau\ \sigma$, which is immediate given that $\hat{k}$ :: `key HK`, i.e. $\hat{k}$ a singleton set, and $\hat{v}' :: \tau$. Both results are obtainable via the induction hypotheses.

In more detail, the interpretation of the set comprehension, $(S1)$, gives the following (11) $k \in \hat{k} \wedge v' \in \hat{v}' \wedge v \in \mathcal{E}_\gamma(k, v') \Longrightarrow v \in \hat{v}$, and (12) $v \in \hat{v} \Longrightarrow \exists k \in \hat{k},\ v' \in \hat{v}' \,.\, v \in \mathcal{E}_\gamma(k, v')$.

Now, $(5, 12)$ give (13) $k \in \hat{k}$, (14) $v' \in \hat{v}'$, and (15) $v \in \mathcal{E}_\gamma(k, v')$ for some k, some v, and some $v \in \hat{v}$.

With this we can apply the induction hypotheses, which give (16) $\hat{k}$ :: `key HK`, and (17) $\hat{v}' :: \tau$.

Now, (16) gives that $\hat{k}$ is singleton, i.e. (18) $\hat{k} = \{k\}$ for k introduced in (13) .

To show $\hat{v} :: \tau$ we must show that $\hat{v}' :: \tau$, $v' \in \hat{v}' \wedge v \in \mathcal{E}_\gamma(k, v') \implies v \in \hat{v}$, and $v \in \hat{v} \implies \exists v' \in \hat{v}' \,.\, v \in \mathcal{E}_\gamma(k, v')$ for the same k, which is possible because of (18). The two former are immediate from (17) and (11) respectively; the latter is immediate from $(12, 18)$, where (18) is crucial for the applicability of (12).

(T-ENC2) Identical to (T-ENC1) with references to high keys replaced by references to low keys.

(T-DEC) We have that $e = \texttt{decrypt}_\gamma\ (e_1, e_2)$, (6) $\Omega \vdash e_1 : \texttt{key}\ \gamma$, and (7) $\Omega \vdash e_2 : \texttt{enc}_\gamma\ \tau\ \sigma$. Now, (4) gives (8) $\langle \hat{M}, e_1 \rangle \Downarrow \hat{k}$, (9) $\langle \hat{M}, e_2 \rangle \Downarrow \hat{u}$, (10) $k \subseteq Key_\gamma$, and (S1) $\hat{v} = \{\mathcal{D}_\gamma(k, u) \mid k \in \hat{k},\ u \in \hat{u}\}$. We want to show that $\hat{v} :: \tau$, which is given from the leaf-determinism of $\hat{k} :: \texttt{key}\ \gamma$, and $\hat{u} :: \texttt{enc}_\gamma\ \tau\ \sigma$ both obtainable from the induction hypotheses. The former guarantees that $\hat{k}$ is a singleton set, and the latter gives $\hat{v}' = \{v \mid u \in \mathcal{E}_\gamma(k, v), u \in \hat{u}\} :: \tau$, for some key k. Now, $\hat{v} \neq \emptyset$ allows us to conclude $\hat{v} :: \tau$, since the fact that decryption with the wrong key fails allows us to establish that $\hat{k} = \{k\}$, and thus $\hat{v} = \hat{v}'$, since they are both the set of values obtained by decrypting $\widehat{u}$ using k.

In more detail, the interpretation of the set comprehension, $(S1)$, gives the following (11) $k \in \hat{k} \wedge u \in \hat{u} \implies \mathcal{D}_\gamma(k, u) \in \hat{v}$, and (12) $v \in \hat{v} \implies \exists k \in \hat{k},\ u \in \hat{u} \,.\, v = \mathcal{D}_\gamma(k, u)$. Now, $(5, 12)$ give (13) $k \in \hat{k}$, (14) $u \in \hat{u}$, and (15) $v = \mathcal{D}_\gamma(k, u)$ for some k, and some u. With this the induction hypotheses are applicable, which give (16) $\hat{k} :: \texttt{key}\ \gamma$, and (17) $\hat{u} :: \texttt{enc}_\gamma\ \tau\ \sigma$. From (16) we get that $\hat{k}$ is singleton, i.e. that (18) $\hat{k} = \{k\}$ for k introduced in (13). From (17) we get that $\{v \mid u \in \mathcal{E}_\gamma(k', v), u \in \widehat{u}\} :: \tau$, for some k'.

Let $\hat{v}' = \{v \mid u \in \mathcal{E}_\gamma(k', v), u \in \widehat{u}\}$; we have that (19) $v' \in \hat{v}' \implies \exists u \in \hat{u} \,.\, u \in \mathcal{E}_\gamma(k', v)$, and (20) $u \in \hat{u} \implies \exists v' \in \hat{v}' \,.\, u \in \mathcal{E}_\gamma(k', v')$.

From $(20, 14)$ we have that $u \in \mathcal{E}_\gamma(k', v')$ for some $v' \in \hat{v}'$. This means that $v' = \mathcal{D}_\gamma(k', u)$; at the same time (15) gives us that $v = \mathcal{D}_\gamma(k, u)$, which allows us to conclude that $k' = k$, since we have assumed that decryption with the wrong key fails.

Now, it remains to be shown that $\hat{v} = \hat{v}'$.

case $v' \in \hat{v}' \implies v' \in \hat{v}$ Assume, (21) $v' \in \hat{v}'$; from $(19, 21)$ we have that (22) $u' \in \hat{u}$, and (23) $u' \in \mathcal{E}_\gamma(k, v')$ for some u'. From $(11, 22)$ we have that $\mathcal{D}_\gamma(k, u') \in \hat{v}$, but $\mathcal{D}_\gamma(k, u') = v'$ from (23), since decryption is left inverse of encryption, and we are done.

case $v \in \hat{v} \implies v \in \hat{v}'$ Assume (21) $v \in \hat{v}$; from $(12, 21)$ we have that (22) $k \in \hat{k}$ (k because of (18)), (23) $u' \in \hat{u}$ for some u', and (24) $v = \mathcal{D}_\gamma(k, u)$. Now, $(23, 20)$ give (25) $u \in \mathcal{E}_\gamma(k, v')$ for some $v' \in \hat{v}'$. From (25) we have that (26) $v' = \mathcal{D}_\gamma(k, u)$, since decryption is left inverse of encryption. Now, since decryption is a deterministic function this means that $v' = v$ and we are done.

□

Before we can define noninterference for expressions, we need to lift the low-equivalence relation to sets. We say that two sets of values are *possibilistically low-equivalent* if for every value in one there exists a low-equivalent one in the other. For values and variable environments this is formulated as follows:

$$\frac{v_1 \in \widehat{v}_1 \Longrightarrow \exists v_2 \in \widehat{v}_2 \,.\, v_1 \sim_\tau v_2 \qquad v_2 \in \widehat{v}_2 \Longrightarrow \exists v_1 \in \widehat{v}_1 \,.\, v_1 \sim_\tau v_2}{\widehat{v}_1 \sim_\tau \widehat{v}_2} \qquad \frac{M_1 \in \widehat{M_1} \Longrightarrow \exists M_2 \in \widehat{M_2} \,.\, M_1 \sim_\Omega M_2 \qquad M_2 \in \widehat{M_2} \Longrightarrow \exists M_1 \in \widehat{M_1} \,.\, M_1 \sim_\Omega M_2}{\widehat{M_1} \sim_\Omega \widehat{M_2}}$$

First, we define noninterference for expressions, which is equivalent to the noninterference of statements defined above. Put simply, if two expressions e_1 and e_2 are run in low-equivalent sets of variable environments, yielding sets of values, then these sets of values should be low-equivalent. More precisely, if an expression e is well-typed in the variable type environment Ω, yielding a result of type τ, then if the expression is successfully run in any pair of low-equivalent variable environments (w.r.t. Ω) then the resulting pair of sets of values should be low-equivalent w.r.t. τ. For expressions this is not enough; noninterference is only provable for leaf deterministic environments. Thus, we demand preservation of low-equivalence only for leaf deterministic sets of variable environments.

$$\begin{aligned} NI(e_1, e_2)_{\Omega,\tau} \equiv \forall \widehat{M_1}, \widehat{M_2} \,.\, \widehat{M_1} :: \Omega \wedge \widehat{M_2} :: \Omega \wedge \widehat{M_1} \sim_\Omega \widehat{M_2} \wedge \\ \langle \widehat{M_1}, e_1 \rangle \Downarrow \hat{v}_1 \wedge \langle \widehat{M_2}, e_2 \rangle \Downarrow \hat{v}_2 \wedge \hat{v}_1 \neq \emptyset \wedge \hat{v}_2 \neq \emptyset \Longrightarrow \widehat{v}_1 \sim_\tau \widehat{v}_2 \end{aligned}$$

With this we can formulate and prove security for well-typed expressions. Because of the set lifted low-equivalence, proving $\widehat{v}_1 \sim_\tau \widehat{v}_2$ amounts to proving $_{(1)}\; v_1 \in \widehat{v}_1 \Longrightarrow \exists v_2 \in \widehat{v}_2 \,.\, v_1 \sim_\tau v_2$, and, similarly, $_{(2)}\; v_2 \in \widehat{v}_2 \Longrightarrow \exists v_1 \in \widehat{v}_1 \,.\, v_1 \sim_\tau v_2$. The proofs of (1) and (2) are symmetric. For brevity we only prove one direction, and only give the proof cases dealing with encryption below; the other cases are completely standard and can be found in the formal proof.

Theorem 7 *Security of expressions (set-lifted version)* $\Omega \vdash e : \tau \Longrightarrow NI(e, e)_{\Omega,\tau}$

Proof. First, let us write out the full theorem and enumerate the assumptions.

$$\begin{aligned} {}_{(1)}\; \Omega \vdash e : \tau \Longrightarrow {}_{(2)}\; \widehat{M_1} :: \Omega \wedge {}_{(3)}\; \widehat{M_2} :: \Omega \wedge {}_{(4)}\; \widehat{M_1} \sim_\Omega \widehat{M_2} \wedge \\ {}_{(5)}\; \langle \widehat{M_1}, e \rangle \Downarrow \hat{v}_1 \wedge {}_{(6)}\; \langle \widehat{M_2}, e \rangle \Downarrow \hat{v}_2 \wedge {}_{(7)}\; \hat{v}_1 \neq \emptyset \wedge {}_{(8)}\; \hat{v}_2 \neq \emptyset \\ \Longrightarrow \widehat{v}_1 \sim_\tau \widehat{v}_2 \end{aligned}$$

Let $i \in \{1, 2\}$ in the following; The proof proceeds by an induction on the type derivation (1), in the form on a case by case analysis on the last type rule applied.

We only consider the rules involving encryption and decryption, and (T-VAR); the latter serves as a representative for the other rules, which are completely standard.

(T-VAR) We have that $e = x$, and (9) $\Omega(x) = \tau$. We want to show that $\forall v_1 \in \hat{v}_1\ \exists v_2 \in \hat{v}_2\ .\ v_1 \sim_\tau v_2$.

Assuming (10) $v_1 \in \hat{v}_1$; $(5, 6)$ give $\hat{v}_i = \{v \mid M \in \hat{M}_i,\ M(x) = v\}$. The interpretation of the set comprehension gives (11$_i$) $\forall v \in \hat{v}_i\ \exists M \in \hat{M}_i\ .\ M(x) = v$, and (12$_i$) $\forall M\ v\ .\ M \in \hat{M}_i \wedge M(x) = v \implies v \in \hat{v}_i$.

Thus, from $(10, 11_1)$ we have that (13) $M_1(x) = v_1$ for some M_1.

From (4) we have that (14) $\forall M_1 \in \hat{M}_1\ \exists M_2 \in \hat{M}_2\ .\ M_1 \sim_\Omega M_2$, which gives (15) $M_2 \in \hat{M}_2$, and (16) $M_1 \sim_\Omega M_2$ for some M_2.

Now, (16) gives (17) $\forall x\ \tau\ .\ \Omega(x) = \tau \implies \exists v_1\ v_2\ .\ M_1(x) = v_1 \wedge M_2(x) = v_2 \wedge v_1 \sim_\tau v_2$

Thus, $(17, 9)$ yield $M_1(x) = v_1$ for v_1 from (11) above, since M_1 is a function, (18) $M_2(x) = v_2$ for some v_2, and (19) $v_1 \sim_\tau v_2$.

Finally, $(12_2, 15, 18)$ allows us to conclude that (20) $v_2 \in \hat{v}_2$, and we are done.

(T-ENC1) We have that $e = \texttt{encrypt}_{\texttt{HK}}\ (e_1, e_2)$, (9) $\Omega \vdash e_1 : \texttt{key HK}$, and (10) $\Omega \vdash e_2 : \tau$. We want to show that $\forall v_1 \in \hat{v}_1\ \exists v_2 \in \hat{v}_2\ .\ v_1 \sim_{\texttt{enc}_{\texttt{HK}}\ \tau\ \texttt{L}} v_2$.

Assuming (11) $v_1 \in \hat{v}_1$; $(5, 6)$ give (12$_i$) $\langle \hat{M}_i, e_1 \rangle \Downarrow \hat{k}_i$, (13$_i$) $\langle \hat{M}_i, e_2 \rangle \Downarrow \hat{v}'_i$, (14$_i$) $\hat{k}_i \subseteq Key_{\texttt{HK}}$, and $\hat{v}_i = \{v \mid k \in \hat{k}_i,\ v' \in \hat{v}'_i,\ v \in \mathcal{E}_\gamma(k, v')\}$. The interpretation of the set comprehension gives (15$_i$) $\forall k \in \hat{k}_i\ v' \in \hat{v}'_i\ .\ v \in \mathcal{E}_{\texttt{HK}}(k, v') \implies v \in \hat{v}_i$, and (16$_i$) $\forall v \in \hat{v}_i \exists v' \in \hat{v}'_i k \in \hat{k}_i\ .\ v \in \mathcal{E}_{\texttt{HK}}(k, v')$.

Now, from $(16_1, 11)$ we have (17) $v'_1 \in \hat{v}'_1$, (18) $k_1 \in \hat{k}_1$, and (19) $v_1 \in \mathcal{E}_{\texttt{HK}}(k_1, v'_1)$. Similarly, (8) ensures the existence values in $\hat{v}_2$, and thus also in $\hat{v}'_2$, and $\hat{k}_2$.

With this the induction hypotheses gives us (20) $\hat{k}_1 \sim_{\texttt{HK}} \hat{k}_2$, and (21) $\hat{v}'_1 \sim_\tau \hat{v}'_2$.

In turn, (20) gives (22) $\forall k_1 \in \hat{k}_1 \exists k_2 \in \hat{k}_2\ .\ k_1 \sim_{\texttt{HK}} k_2$, and (21) gives (23) $\forall v'_1 \in \hat{v}'_1 \exists v'_2 \in \hat{v}'_2\ .\ v'_1 \sim_\tau v'_2$.

Thus, $(17, 23)$ give (24) $v'_2 \in \hat{v}'_2$ for some v'_2, and (25) $v'_1 \sim_\tau v'_2$. Similarly, $(18, 22)$ give (26) $k_2 \in \hat{k}_2$ for some k_2, and (27) $k_1 \sim_{\texttt{HK}} k_2$.

Now, Equation 1 from Section 3.3 together with (19) give us (28) $v_2 \in \mathcal{E}_{\texttt{HK}}(k_2, v'_2)$ for some v_2 such that (29) $v_1 \doteq v_2$.

From $(15_2, 26, 24, 28)$ we get (30) $v_2 \in \hat{v}_2$, and from the fact that decryption is a left inverse of encryption together with $(29, 27, 25, 28, 19)$, and (LE-ENC-L1) allows us to draw the conclusion that $v_1 \sim_{\texttt{enc}_{\texttt{HK}}\ \tau\ \texttt{L}} v_2$, and we are done.

(T-ENC2) We have that $e = \texttt{encrypt}_{\texttt{HK}}\ (e_1, e_2)$, (9) $\Omega \vdash e_1 : \texttt{key LK}$, and (10) $\Omega \vdash e_2 : \tau$. We want to show that $\forall v_1 \in \hat{v}_1\ \exists v_2 \in \hat{v}_2\ .\ v_1 \sim_{\texttt{enc}_{\texttt{LK}}\ \tau\ \sigma} v_2$.

The proof for (T-ENC2) is similar to the proof for (T-ENC1). The only difference is that we proceed with a case on σ.

case $\sigma = \texttt{L}$

The proof is identical to proof of (T-ENC1), with all lemmas regarding secret encryption replaced by their public encryption equivalents (Equation 1 from Section 3.3 does not have a public encryption equivalent — there is no need since (LE-ENC-L2) does not demand $v_1 \doteq v_2$).

case $\sigma = \mathtt{H}$

The result is immediate from the existence of $v_2 \in \hat{v}_2$ established in the above case, and (LE-ENC-H).

(T-DEC) We have that $e = \mathtt{decrypt}_\gamma\ (e_1, e_2)$, (9) $\Omega \vdash e_1 : \mathtt{key}\ \gamma$,and (10) $\Omega \vdash e_2 : \mathtt{enc}_\gamma\ \tau\ \sigma$. We want to show that $\forall v_1 \in \hat{v}_1\ \exists v_2 \in \hat{v}_2\ .\ v_1 \sim_{\tau^\sigma} v_2$.

Assuming (10) $v_1 \in \hat{v}_1$; $(5, 6)$ give (12_i) $\langle \hat{M}_i, e_1\rangle \Downarrow \hat{k}_i$, ($13_i$) $\langle \hat{M}_i, e_2\rangle \Downarrow \hat{v}'_i$, ($14_i$) $\hat{k} \subseteq Key_\gamma$, and $\hat{v}_i = \{\mathcal{D}_\gamma(k, v') \mid k \in \hat{k}_i,\ v' \in \hat{v}'_i\}$. The interpretation of the set comprehension gives (15_i) $\forall k_i \in \hat{k}_i\ v'_i \in \hat{v}'_i\ .\ v_i = \mathcal{D}_\gamma(k_i, v'_i) \implies v_i \in \hat{v}_i$, and ($16_i$) $\forall v_i \in \hat{v}_i \exists v'_i \in \hat{v}'_i,\ k_i \in \hat{k}_i\ .\ v_i = \mathcal{D}_\gamma(k_i, v'_i)$.

Now, $(16_1, 10)$ give us that (17) $v'_1 \in \hat{v}'_1$, (18) $k_1 \in \hat{k}_1$, and (19) $v_1 = \mathcal{D}_\gamma(k_1, v'_1)$.

From (8) we get the existence of $v_3 \in \hat{v}_2$, which via (16_2) gives (20) $v'_3 \in \hat{v}'_2$, (21) $k_2 \in \hat{k}_2$, and (22) $v_3 = \mathcal{D}_\gamma(k_2, v'_3)$.

From the induction hypotheses we can now show that (23) $\hat{k}_1 \sim_\gamma \hat{k}_2$, and (24) $\hat{v}'_1 \sim_{\mathtt{enc}_\gamma\ \tau\ \sigma} \hat{v}'_2$.

Furhter, $(23, 24)$ give (25) $\forall k_1 \in \hat{k}_1\ \exists k_2 \in \hat{k}_2\ .\ k_1 \sim_\gamma k_2$, and (26) $\forall v'_1 \in \hat{v}'_1\ \exists v'_2 \in \hat{v}'_2\ .\ v'_1 \sim_{\mathtt{enc}_\gamma\ \tau\sigma} v'_2$.

Theorem 6 gives that (27_i) $\hat{k}_i :: \gamma$, which, in turn, yields that $\hat{k}_i$ are singleton sets, i.e. (28_i) $\hat{k}_i = \{k_i\}$, for k_i introduced in $(18, 21)$. Thus, $(25, 18)$ give (29) $k_2 \in \hat{k}_2$, and (30) $k_1 \sim_\gamma k_2$; $(26, 17)$ gives (31) $v'_2 \in \hat{v}'_2$, and (32) $v'_1 \sim_{\mathtt{enc}_\gamma\ \tau\sigma} v'_2$.

We proceed by a case analysis of the key level γ and σ.

case $\gamma = \mathtt{LK}$

We have (30), i.e. $k_1 \sim_{\mathtt{LK}} k_2$ gives that $k_1 = k_2$ and, thus, that (33) $\hat{k}_1 = \hat{k}_2$ by (28_i).

case $\sigma = \mathtt{L}$

Now, (32), i.e. $v'_1 \sim_{\mathtt{enc}_{\mathtt{LK}}\ \tau\mathtt{L}} v'_2$, gives ($34_i$) $v''_i = \mathcal{D}_{\mathtt{LK}}(k'_i, v'_i)$, for some k'_i,and some v''_i, (35) $v''_1 \sim_{tolow(\tau)} v''_2$.

From (28_i) $k'_i = k_i$, from (19) and since decryption is a deterministic function $v''_1 = v_1$; let v''_2 be denoted v_2 in the following. Now, $v_2 \in \hat{v}_2$ by $(15_2, 29, 31, 34_2)$, and $v_1 \sim_{\tau^{\mathtt{L}}} v_2 \equiv v_1 \sim_\tau v_2$, which is immediate from (35).

case $\sigma = \mathtt{H}$

(8) gives us $v_2 \in \hat{v}_2$ for some v_2; $v_1 \sim_{\tau^{\mathtt{H}}} v_2$ is easily proven for any v_i.

case $\gamma = \mathtt{HK}$

We have (32), i.e. $v'_1 \sim_{\mathtt{enc}_{\mathtt{HK}}\ \tau\mathtt{L}} v'_2$, gives ($34_i$) $v''_i = \mathcal{D}_{\mathtt{LK}}(k'_i, v'_i)$, for some k'_i,and some v''_i, (35) $v''_1 \sim_\tau v''_2$.

From (28_i) $k'_i = k_i$, from (19) and since decryption is a deterministic function $v''_1 = v_1$; let v''_2 be denoted v_2 in the following. Now, $v_2 \in \hat{v}_2$ by $(15_2, 29, 31, 34_2)$, and we are done.

□

With this we are ready to state and prove the top-level security theorem for expressions corresponding to the top-level security statement for commands stated previously in Section 3.3

Theorem 8 *Security of expressions (top-level version)*

$$\text{(1) } \Omega \vdash e : \tau \Longrightarrow \text{(2) } M_1 : \Omega \wedge \text{(3) } M_2 : \Omega \wedge \text{(4) } M_1 \sim_\Omega M_2 \wedge$$
$$\text{(5) } \langle M_1, e\rangle \Downarrow \hat{v}_1 \wedge \text{(6) } \langle M_2, e\rangle \Downarrow \hat{v}_2 \wedge \text{(7) } \hat{v}_1 \neq \emptyset \wedge \text{(8) } \hat{v}_2 \neq \emptyset \Longrightarrow \hat{v}_1 \sim_\tau \hat{v}_2$$

Proof. In the following, let $i \in \{1, 2\}$. Again, we only show one direction; the other direction is symmetric. Thus, we must show that $v_1 \in \widehat{v}_1 \Longrightarrow \exists v_2 \in \widehat{v}_2 \,.\, v_1 \sim_\tau v_2$.

Assume (9) $v_1 \in \widehat{v}_1$; (8) gives the existence of (10) $v_2 \in \widehat{v}_2$. Now, $(5, 6)$ gives (11$_i$) $\forall v \,.\, \langle M_i, e\rangle \Downarrow v \iff v \in \hat{v}_i$. Now, $(9, 10, 11_i)$ gives $\langle M_i, e\rangle \Downarrow v_i$, and Lemma 2 gives the existence of $\hat{v}'_i$ such that (12$_i$) $\langle\{M_i\}, e\rangle \Downarrow \hat{v}'_i$ and (13$_i$) $v_i \in \hat{v}'_i$. Noting that by Lemma 5 we have that $\{M_i\} :: \Omega$ from $M_i : \Omega$, Theorem 7 becomes applicable and gives (14) $\widehat{v}'_1 \sim_\tau \widehat{v}'_2$. Now, (14) gives $v_1 \in \widehat{v}'_1 \Longrightarrow \exists v_2 \in \widehat{v}'_2 \,.\, v_1 \sim_\tau v_2$, and, thus, there exists v_2 such that (15) $v_1 \sim_\tau v_2$. Now, from Lemma 3 and (13_2) we get that (16) $\langle\{M_2\}, e\rangle \Downarrow v_2$.

With this we are done; we have already proven $v_1 \sim_\tau v_2$ (15) and $v_2 \in \widehat{v}_2$ is immediate from (11_2). □

For commands, the corresponding top-level security theorem is as follows. Since the commands do not include anything specific to encryption the proof is straightforward.

Theorem 9 *Security of commands (top-level version)*

$$\text{(1) } \Sigma_1, \mathtt{L} \vdash c \wedge \text{(2) } E_1 : \Sigma \wedge \text{(3) } E_2 : \Sigma \wedge \text{(4) } E_1 \sim_\Sigma E_2 \wedge$$
$$\text{(5) } \langle E_1, c\rangle \Downarrow \hat{E}'_1 \wedge \text{(6) } \langle E_2, c\rangle \Downarrow \hat{E}'_2 \wedge \text{(7) } \hat{E}'_1 \neq \emptyset \wedge \text{(8) } \hat{E}'_2 \neq \emptyset \Longrightarrow \hat{E}'_1 \sim_\Sigma \hat{E}'_2$$

where $\hat{E}_1 \sim_\Sigma \hat{E}_2$ is the immediate set-lifted structural extension of low-equivalence to environments, and $E : \Sigma$ is the structural extension of well-formedness to environments.

Proof. By induction on the derivation of $\Sigma_1, \mathtt{L} \vdash c$. □

6 Extensions

In this section we consider two extensions: integrity and public-key cryptography.

6.1 Integrity

Confidentiality classifies information into public (low-confidentiality) and secret (high-confidentiality), i.e., information that may or may not be given to the world, respectively. Dually, integrity classifies information into *untrusted* (or *low-integrity*) and *trusted* (or *high-integrity*), i.e., whether the information may or may not have been *affected* by the world.

Tracking the integrity of data enables us to explore some additional dimensions of cryptography: weaknesses of the encryption algorithms and the effect of encryption on integrity. Consider for example, a primitive that is vulnerable to chosen ciphertext attacks. With integrity controls, it is natural to express the restriction that untrusted encrypted values may not be decrypted.

In the presence of integrity the security levels for values are pairs of the form (σ, ι), where σ is a confidentiality level, and ι is a corresponding integrity level. The following tables define two functions—$\mathtt{safe}_{\mathcal{E}}(\alpha, (\sigma, \iota))$ and $\mathtt{safe}_{\mathcal{D}}(\alpha, (\sigma, \iota))$—that indicate if it is safe to encrypt (decrypt) a plaintext (ciphertext) of security level (σ, ι) with an encryption scheme that has property α. Here α ranges over standard notions [5]—IND-CCA (indistinguishable under chosen-ciphertext attacks) and IND-CPA (indistinguishable under chosen-plaintext attacks).

	(H,H)	(L,L)	(H,L)	(L,H)
IND-CCA	safe	safe	safe	safe
IND-CPA	safe	safe	safe	safe

$\mathtt{safe}_{\mathcal{E}}(\alpha, (\sigma, \iota))$

	(H,H)	(L,L)	(H,L)	(L,H)
IND-CCA	safe	safe	safe	safe
IND-CPA	safe	-	-	safe

$\mathtt{safe}_{\mathcal{D}}(\alpha, (\sigma, \iota))$

In this way we can provide different type rules for different assumptions on the vulnerability properties of the encryption and decryption algorithms:

$$\text{(T-ENC*)}\ \frac{\Omega \vdash e_1 : \mathtt{key\ HK} \quad \Omega \vdash e_2 : \tau \quad \mathit{lvl}(\tau) = (\sigma, \iota) \quad \mathtt{safe}_{\mathcal{E}}(\alpha, (\sigma, \iota))}{\Omega \vdash \mathtt{encrypt}^{\alpha}_{\mathtt{HK}}\ (e_1, e_2) : \mathtt{enc}_{\mathtt{HK}}\ \tau\ (\mathtt{L}, \mathtt{H})}$$

$$\text{(T-DEC*)}\ \frac{\Omega \vdash e_1 : \mathtt{key}\ \gamma \quad \mathtt{safe}_{\mathcal{D}}(\alpha, (\sigma, \iota)) \quad \Omega \vdash e_2 : \mathtt{enc}_{\gamma}\ \tau\ (\sigma, \iota)}{\Omega \vdash \mathtt{decrypt}^{\alpha}_{\gamma}\ (e_1, e_2) : \tau^{(\sigma,\iota)}}$$

6.2 A note on the integrity of keys

The current model allows very limited interaction with keys apart from encryption. Since the values of keys cannot be programmatically inspected, the power of the attacker is limited to choose between secure keys. Thus, the model cannot in its present form distinguish between encryption with high and low-integrity keys w.r.t. *confidentiality*. The intuition is clear: since the attacker can only choose between secure keys, that choice will give different but safe encrypted values.

6.3 Public-key cryptography

Even though the present system deals only with symmetric-key cryptography, there is nothing in the model that prevents modeling public-key cryptography. The set of high keys would contain the *private* keys and the set of low keys would contain the *public* keys, where the private keys and the public keys are dual. In this system values encrypted with public keys would be considered low, since only actors with access to the private keys would be able to decrypt them.

However, public-key cryptography is most interesting in the presence of integrity. In the same way we can model that encryption of secrets using secret keys results in low values, we can model that encryption raises the integrity of the encrypted value to the integrity of the key, which corresponds to signing.

7 Programming with encryption: Examples

We have implemented a prototype of the type system and mechanically type-checked two applications: secure backup and a Wide-Mouthed-Frog protocol implementation. In both examples the type system prevents dangerous insecurities such as sending sensitive unencrypted data over a low channel or not using a secret key for encryption. This section discusses some interesting fragments of these implementations. [2]

7.1 Secure data backup

In the secure backup scenario a low-confidentiality channel is used for sending sensitive information to the remote storage. Listing 2 presents the code for the backup operation. Here and below we aid the reader by providing explicit variable type declarations.

```
K key HK;
backup enc HK (int H) L;

actor Backup {
  data int H;
  ctxt enc HK (int H) L;
  data := ...
  ctxt := encrypt_HK(K, data);
  out backup ctxt;
}
```

Listing 2. Backup code

We declare high key `K` and low channel `backup`. The type of the latter says that only encrypted high integers may be sent over this channel.

[2] In the examples below some variables and channels that are shared between actors are declared before the code for actors. This is done to avoid double declarations and the prototype implementation uses these global declarations when building local environments of every actor.

Lines 5 and 7 declare and initialize a high integer variable `data`. Line 6 declares the variable `ctxt` of type `enc HK (int H) L`. On line 8 the value of variable `data` is encrypted with high key `K` and the resulting ciphertext is assigned to the variable `ctxt`. Since type of `ctxt` matches the type of the `backup` channel it might be sent over this channel. This is done by the `out` command on line 9.

When recovering data, an actor reads the data from the low channel and decrypts it. Assuming the same global declarations Listing 3 presents the recovery code. Here, line 4 reads data from the `backup` channel. It's decrypted using the key `K` on line 5.

```
1 actor Restore {
2   data int H;
3   ctxt enc HK (int H) L;
4   in ctxt backup;
5   data := decrypt_HK(K, ctxt);
6 }
```

Listing 3. Recovery code

An example of an easy-to-overlook error is to have the following line in place of line 9 in the body of actor `Backup`: `out backup data;`. This is an insecurity that the type system rejects. Generally, in the secure backup example the type system ensures that secret data is encrypted before it is sent over the `backup` channel, thus preventing accidental leaks.

7.2 Wide-Mouthed-Frog protocol

The Wide-Mouthed-Frog protocol [8] is a simple key exchange protocol with trusted server and timestamps. In this protocol secret keys K_{AS} and K_{BS} are shared between server S and principals A and B, respectively. Principal A generates a fresh session key K_{AB}, which is transferred to B in two messages:

$$1.\ A \rightarrow S : A, \{T_A, B, K_{AB}\}K_{AS}$$

$$2.\ S \rightarrow B : \{T_S, A, K_{AB}\}K_{BS}$$

The first message consists of A's name and a tuple encrypted with the shared key K_{AS}. This tuple contains three elements—a timestamp T_A, the name of principal B, and a generated key K_{AB}. Upon receipt of this message, S decrypts it, checks the timestamp, replaces T_A with its own timestamp T_S, encrypts it with key K_{BS}, and forwards the resulting message to B. Principal B then checks whether the second message is timely.

Obviously, there is more to implementation of the protocol than expressed by the two-step description. Our type system guarantees that implementations do not introduce information-flow leaks in the protocol. Listing 4 presents the implementation of this protocol for principal A.

This program declares two channels: `chanS` for communicating with the server,

```
 1 Kas key HK;
 2 chanS <int L, enc HK (<int L, <int L, key HK>>) L>;
 3 chanAB enc HK (int H) L;
 4 actor A {
 5   idA int L; idB int L; tsA int L;
 6   messageToB int H;
 7   Kab key HK;
 8   //  ... initialization
 9   newkey (Kab, HK);
10   out chanS <idA, encrypt_HK(Kas, <tsA,<idB, Kab>>)>;
11   out chanAB encrypt_HK (Kab, messageToB);
12 }
```

Listing 4. WMF Implementation

and `chanAB` for sending messages to B, once the key has been exchanged. The type of the channel `chanS` corresponds to the first message in the protocol—a pair consisting of a low integer and an encryption with high key of a three-element tuple (expressed by nested pairs). Since the level of the key used for encrypting this tuple is `high`, it is safe to label the result of encryption as `low`. The body of the `actor` declaration defines low-confidentiality variables `idA` and `idB` that stand for the names of the principals; variable `tsA` stores the current timestamp; the high-confidentiality variable `messageToB` contains the information that A wants to send to B.

The new key is generated on line 9. Line 10 constructs the first message of the protocol and sends it to the server. Line 11 uses the newly generated key and sends the secret message to the principal B.

The following listing presents the code for the actors `Server` and `B` in the Wide-mouthed-frog protocol implementation. Here we assume existence of appropriate macros `IS_FRESH` and `GET_TIME`.

```
Kas key HK;  // declaration of
Kbs key HK;  // shared keys

// channel for accepting requests
chanS <int L, enc HK (<int L, <int L, key HK>>) L  >;

// channels for communicating with principals
chanA enc HK ( <int L, <int L, key HK>> ) L;
chanB enc HK ( <int L, <int L, key HK>> ) L;

actor S {
    idA int L;  idB int L;

    tsS int L; // time stamp
    ... // initialize identifiers

    while (1) {
      idFrom  int L;   idTo    int L;    tsFrom  int L;    Kab     key HK;
      request <int L, enc HK (<int L, <int L, key HK>>) L>;
      msg     <int L, <int L, key HK>>;

      in request chanS; // accepting a request from a principal
      idFrom := fst (request);

      if idFrom == idA then {
        msg := decrypt_HK(Kas, snd (request));
      } else {
        msg := decrypt_HK(Kbs, snd (request));
      };

      tS := GET_TIME
      tsFrom := fst(msg);
      if IS_FRESH(tsFrom, tsS) then {
        idTo := fst (snd(msg));
        Kab  := snd (snd(msg));

        // forwarding the request to the other principal
        if idTo == idA then {
          out chanA encrypt_HK(Kas, <tsS, <idFrom, Kab>>);
        } else {if idTo == idB then {
          out chanB encrypt_HK(Kbs, <tsS, <idFrom, Kab>>);
        } else {};};
      } else {} ;
   };
}

actor B {
    ctxt enc HK ( <int L, <int L, key HK>> ) L;
    msg  <int L, <int L, key HK>>;
    Kab key HK; idFrom int L;
    tsS int L;    tsB int L;
    ... // initialize identifiers and timestamp
    in ctxt chanB;
    msg := decrypt_HK (Kbs, ctxt);
    tsS := fst (msg);
    if IS_FRESH(tsS, tsB) then {
       idFrom := fst (snd (msg));
       Kab    := snd (snd(msg));

       //  get a message from A
       cmsg enc HK (int H) L;
       in cmsg chanAB;
       messageFromA int H;
       messageFromA := decrypt_HK(Kab, cmsg);
    } else {};
}
```

In this example, the type system prevents non-secret session keys in the key establishment protocol. As in the previous example, it also guarantees that secret information may not leave the system unless it is encrypted with a secret key.

8 Related work

As mentioned in the introduction, declassification models are sometimes used to justify cryptographic primitives in languages with information-flow control. Declassification mechanisms facilitate information release. A recent classification of declassification [32] suggests that information release policies represent aspects of *what* is declassified, by *whom*, *when* and *where* in the system. These correspond to dimensions of information release. The relation of our model to declassification is somewhat subtle, because the goal of masking is information hiding rather than information release.

Furthermore, attempts at framing cryptographically-masked flows into different dimensions do not always lead to satisfactory results. For example, releasing the difference between two values of a secret whenever the results of its encryption are different can be a deceptive policy when assumptions about the underlying cryptographic primitives are not explicitly stated. If the underlying encryption function is bijective (assuming the key is fixed) then releasing the result of encryption is equivalent to releasing the secret itself. This phenomenon applies to typical policies from the *what* dimension, such as delimited release [31].

Another example of releasing the secret itself, together with the result of a cryptographic primitive applied to the secret, can be found in [9]. The password checker example is based on matching the hash of the password with the hash of a user query. The password has a label $H \overset{cert}{\rightsquigarrow} L$, which means that the level of the password is eventually declassified from high to low. This, however, allows the password itself to be released to the attacker in cleartext.

Nevertheless, declassification is meaningful in the context of cryptographic computation when the attacker is capable of learning some information from ciphertext. Temporal policies express *when*, at earliest, the attacker might learn the secret. Volpano and Smith's relative secrecy [37,36] guarantees that the attacker cannot learn the secret in polynomial time in the size of the secret. Approaches by Laud [20,21], Laud and Vene [23], provide computational guarantees for a simple imperative language but with the assumption that keys can be statically distinguished. Smith and Alpizar [33] present a type system for a language with random assignment, encryption, and decryption and establish computational security for typable programs.

As mentioned previously, Laud's recent work [22] adopts this article as a starting point and bridges cryptographically-masked flows with computational security. In fact, Laud formally proves a conjecture from an earlier version [2] of this article that SEM-CPA and INT-PTXT properties of underlying cryptographic primitives (extended to hide key identities) are sufficient to guarantee computational security for programs that satisfy our possibilistic noninterference.

Mitchell et al. [24,26] reason about security with respect to polynomial-time attackers for a form of the π calculus.

A source of our inspiration is Abadi's secrecy model for symmetric-key cryptographic protocols [1]. This model assumes that an attacker is unable to decrypt ciphertexts encrypted with secret keys. Compared to [1], we end up with simpler typing rules. For example, because of the probabilistic encryption assumption, we do not need to deal with explicit confounders. In addition, our approach accommodates natural extensions with integrity and public-key cryptography. Another source of inspiration is a logical relations technique by Sumii and Pierce that facilitates manual security proofs for cryptographic protocols [34]. This technique is not accompanied by static enforcement mechanisms (such as a type system), however.

Gordon and Jeffrey [18] extend Abadi's work to multiple security levels that may be dynamically created and may become compromised. This and other work within Gordon and Jeffrey's Cryptyc project, however, relies on trace-based properties (such as correspondence) that are weaker than noninterference. Dam and Giambiagi's work on *admissibility* [13,16] focuses on protocol implementation, with the goal that information leaks in the implementation must adhere to those declared in protocol specification.

Duggan's and Chothia et al.'s cryptographic types [15,10] help enforce security for a distributed programming language. This is realized through a combination of static and dynamic checks, leading to access-control guarantees (albeit without information-flow guarantees) for secrecy and integrity. Myers et al.'s qualified robustness [28] is based on a possibilistic treatment of *endorsement*, operation dual to declassification.

Hicks et al. [19] define a notion of *noninterference modulo trusted functions*, which requires parts of programs free of cryptographic functions to be in a certain sense indistinguishable. The cryptographic functions are trusted to release information if their security labels satisfy trust constraints. It is a worthwhile direction for future work to formally investigate the relation to *noninterference modulo trusted functions*. We do not expect it to be straightforward because the definition of the indistinguishability relation from [19] involves two-level semantics.

Vaughan and Zdancewic [35] present a language in which security labels are connected to public-key cryptography. Based on the decentralized label model [27], they explore rich confidentiality and integrity policies. However, their semantic security condition appears to relate all ciphertexts as indistinguishable, which may result in occlusion (cf. Section 3.2).

Finally, the first and last authors have proposed a *gradual release* framework [4] that unifies revelation-based and encryption-based policies. This framework conservatively extends cryptographically-masked flows with possibilities of reasoning about key release, and offers a type-based enforcement mechanism that prevents premature key release.

9 Conclusions

We have developed an approach to tracking information flow in the presence of cryptographic operations, based on possibilistic noninterference. We have argued that a possibilistic treatment of cryptographic operations leads to a natural model of attackers that may not learn useful information from ciphertexts. This model has a close connection to probabilistic encryption and it naturally connects to computational adversary models [22].

Our case for possibilistic noninterference is driven by the possibility of capitalizing on the available machinery for reasoning about noninterference in programming languages. We have demonstrated that possibilistic noninterference can be provably and straightforwardly enforced via a security-type system for a language that includes cryptographic primitives and message passing. We have formalized the main proof of soundness in the proof assistant Coq. The type system is amenable to extensions, including integrity and public-key cryptography, which makes it attractive for developing secure implementations of non-trivial cryptographic protocols.

Acknowledgments We wish to thank Martín Abadi, Peeter Laud, David Sands, and the anonymous reviewers for helpful comments. This work was supported by the Swedish research agencies SSF, Vinnova, VR, and by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project.

References

[1] M. Abadi. Secrecy by typing in security protocols. *J. ACM*, 46(5):749–786, September 1999.

[2] A. Askarov, D. Hedin, and A. Sabelfeld. Cryptographically-masked flows. In *Proc. Symp. on Static Analysis*, LNCS, pages 353–369. Springer-Verlag, August 2006.

[3] A. Askarov and A. Sabelfeld. Security-typed languages for implementation of cryptographic protocols: A case study. In *Proc. European Symp. on Research in Computer Security*, volume 3679 of *LNCS*, pages 197–221. Springer-Verlag, September 2005.

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

[5] M. Bellare, A. Desa, D. Pointcheval, and P. Rogaway. Relations among notions of security for public-key encryption schemes. In *Advances in Cryptology- Crypto 98*, volume 1462 of *LNCS*, pages 26–46, January 1998.

[6] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *ACM Conference on Computer and Communications Security*, pages 62–73, November 1993.

[7] J. Black, P. Rogaway, and T. Shrimpton. Encryption-scheme security in the presence of key-dependent messages. In *Selected Areas in Cryptography*, volume 2595 of *LNCS*, pages 62–75. Springer-Verlag, August 2002.

[8] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, February 1990.

[9] S. Chong and A. C. Myers. Security policies for downgrading. In *ACM Conference on Computer and Communications Security*, pages 198–209, October 2004.

[10] T. Chothia, D. Duggan, and J. Vitek. Type-based distributed access control. In *Proc. IEEE Computer Security Foundations Workshop*, pages 170–186, 2003.

[11] D. Clark and S. Hunt. Observation, nondeterminism and nondeducability on strategies. Workshop presentation at PLID'07, 3rd International Workshop on Programming Language Dependence and Independence, August 2007.

[12] E. S. Cohen. Information transmission in sequential programs. In R. A. DeMillo, D. P. Dobkin, A. K. Jones, and R. J. Lipton, editors, *Foundations of Secure Computation*, pages 297–335. Academic Press, 1978.

[13] M. Dam and P. Giambiagi. Confidentiality for mobile code: The case of a simple payment protocol. In *Proc. IEEE Computer Security Foundations Workshop*, pages 233–244, July 2000.

[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] D. Duggan. Cryptographic types. In *Proc. IEEE Computer Security Foundations Workshop*, pages 238–252, June 2002.

[16] P. Giambiagi and M. Dam. On the secure implementation of security protocols. In *Proc. European Symp. on Programming*, volume 2618 of *LNCS*, pages 144–158. Springer-Verlag, April 2003.

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

[18] A. Gordon and A. Jeffrey. Secrecy despite compromise: Types, cryptography, and the pi-calculus. In *Proc. CONCUR'05*, number 3653 in LNCS, pages 186–201. Springer-Verlag, August 2005.

[19] B. Hicks, D. King, and P. McDaniel. Declassification with cryptographic functions in a security-typed language. Technical Report NAS-TR-0004-2005, Network and Security Center, Department of Computer Science, Pennsylvania State University, May 2005.

[20] P. Laud. Semantics and program analysis of computationally secure information flow. In *Proc. European Symp. on Programming*, volume 2028 of *LNCS*, pages 77–91. Springer-Verlag, April 2001.

[21] P. Laud. Handling encryption in an analysis for secure information flow. In *Proc. European Symp. on Programming*, volume 2618 of *LNCS*, pages 159–173. Springer-Verlag, April 2003.

[22] P. Laud. On the computational soundness of cryptographically masked flows. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 337–348, January 2008.

[23] P. Laud and V. Vene. A type system for computationally secure information flow. In *Proc. Fundamentals of Computation Theory*, volume 3623 of *LNCS*, pages 365–377, August 2005.

[24] P. Lincoln, J. C. Mitchell, M. Mitchell, and A. Scedrov. A probabilistic poly-time framework for protocol analysis. In *ACM Conference on Computer and Communications Security*, pages 112–121, November 1998.

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

[26] J. C. Mitchell. Probabilistic polynomial-time process calculus and security protocol analysis. In *Proc. European Symp. on Programming*, volume 2028 of *LNCS*, pages 23–29. Springer-Verlag, April 2001.

[27] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proc. ACM Symp. on Operating System Principles*, pages 129–142, October 1997.

[28] A. C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification and qualified robustness. *J. Computer Security*, 14(2):157–196, May 2006.

[29] National Institute of Standards and Technology. Special publication 800-38a. Recommendation for block cipher modes of operation, December 2001.

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

[31] A. Sabelfeld and A. C. Myers. A model for delimited information release. In *Proc. International Symp. on Software Security (ISSS'03)*, volume 3233 of *LNCS*, pages 174–191. Springer-Verlag, October 2004.

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

[33] G. Smith and R. Alpizar. Secure information flow with random assignment and encryption. In *Proc. 4th ACM Workshop on Formal Methods in Security Engineering*, pages 33–43, November 2006.

[34] E. Sumii and B. Pierce. Logical relations for encryption. In *Proc. IEEE Computer Security Foundations Workshop*, pages 256–269, June 2001.

[35] J. Vaughan and S. Zdancewic. The cryptographic decentralized label model. In *Proc. IEEE Symp. on Security and Privacy*, pages 192–206, May 2007.

[36] D. Volpano. Secure introduction of one-way functions. In *Proc. IEEE Computer Security Foundations Workshop*, pages 246–254, July 2000.

[37] D. Volpano and G. Smith. Verifying secrets and relative secrecy. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 268–276, January 2000.

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