Cryptographically-Masked Flows

Aslan Askarov Daniel Hedin Andrei Sabelfeld

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

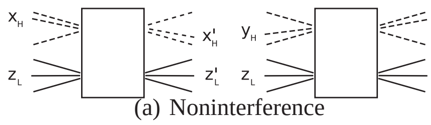
Abstract. Cryptographic operations are essential for many security-critical systems. Reasoning about information flow in such systems is challenging because typical (noninterference-based) information-flow definitions allow no flow from secret to public data. Unfortunately, this implies that programs 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 paper 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

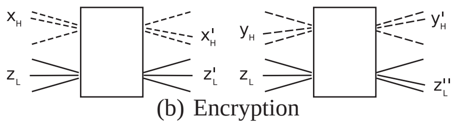
Cryptographic operations are ubiquitous in security-critical systems. Reasoning about information flow in such systems is challenging because typical information-flow definitions allow no flow from secret to public data. The latter requirement underlies *noninterference* [11, 16], which demands that 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 [29, 4]. Attempts at framing cryptographically-masked flows into different dimensions have been made although, as we discuss, not always with satisfactory results.

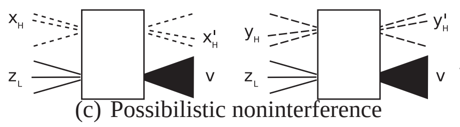
In this paper, we introduce cryptographic primitives into an information-flow setting while preserving a form of noninterference property. This is achieved by building



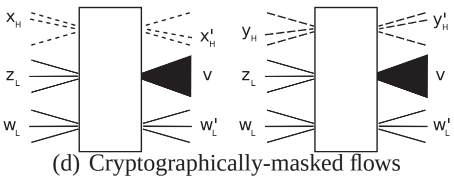
(a) Noninterference



(b) Encryption



(c) Possibilistic noninterference



(d) Cryptographically-masked flows

Fig. 1. From noninterference to cryptographically-masked flows

in 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 x_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 $z = \texttt{enc}(k, x)$ 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* [24] (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 paper makes a case for possibilistic noninterference as a natural model for cryptographically-masked flows. Further, we 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.

sec. levels	$\sigma ::= \mathtt{L} \mid \mathtt{H}$	basic types	$t ::= \mathtt{int} \mid \mathtt{enc}_\gamma\ \tau$
key levels	$\gamma ::= \mathtt{P} \mid \mathtt{S}$	prim. types	$\tau ::= t\ \sigma \mid \mathtt{key}\ \gamma \mid (\tau_1, \tau_2)$
global decls.	$gd ::= \mathtt{global}\ x\ \gamma \mid ch\ \tau$	local decls.	$ld ::= x\ \tau$

expressions $e ::= n \mid x \mid e_1\ op\ e_2 \mid \mathtt{enc}_\gamma\ (e_1, e_2) \mid \mathtt{dec}_\gamma\ (e_1, e_2) \mid \mathtt{newkey}\ \gamma \mid (e_1, e_2)$
$\quad \mid\ \mathtt{fst}(e) \mid \mathtt{snd}(e)$

statements $c ::= \mathtt{skip} \mid x := e \mid \mathtt{if}\ e\ \mathtt{then}\ b_1\ \mathtt{else}\ b_2 \mid \mathtt{while}\ e\ \mathtt{do}\ b \mid \mathtt{out}(ch, e)$
$\quad \mid\ \mathtt{in}(x, ch)$

block $b ::= \{ld_1; \ldots ld_n; c_1; \ldots; c_m\}$

actor $actor ::= A\ b$ program $prog ::= gd_1; \ldots gd_n; actor_1 \ldots actor_m$

Fig. 2. Syntax

2 Language

We explore how to model cryptographic flows in a small imperative language equipped with primitive encryption functions, dynamic key generation, and channels for communication. This section introduces the syntax and semantics of the language. For space reasons we are forced to omit the standard features of the language. The complete rules can be, however, found in the full version of this paper [3].

Syntax The syntax of the language is defined in Figure 2. Let $x \in \mathit{VarName}$ range over the set of variable names and $ch \in \mathit{ChanName}$ range over the set of channel names. A *program* consists of a sequence of *global declarations* followed by a sequence of *actors*. A global declaration is either a declaration of a *global key* or the declaration of a channel. Global keys are declared by associating a variable name with a *key level*. Values and keys have corresponding *security levels*. Values are either *public (low)* L or *secret (high)* H. The key levels declare the maximum value security level the key can safely encrypt. In particular, a key of level S may safely encrypt public and secret values, whereas a key of level P may only safely encrypt public values. Let $\mathit{KeyLvl} = \{\mathtt{S}, \mathtt{P}\}$ be the set of key levels. Global keys are assumed to have appropriate values at the beginning of the execution of a program and correspond to initial shared secrets between the actors of the program. A *channel* is declared by associating a channel name with the type of the messages that will be sent over the channel. Let A range over the set of *actor names*. An actor is defined by naming a *block*, representing the code of the actor. A block is simply a sequence of variable declarations followed by a sequence of commands. Variables are local to the block in which they are declared. The commands include the standard commands of an imperative language and commands for sending on and receiving from a given channel. Apart from expressions for generating new keys and for encryption and decryption, expressions are standard: integers, variables, total binary operators, pair formation, and projection.

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 the declared 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. [27]). First we define the values and environments, which are used in the following definition of the semantics of expressions and commands. Let $n \in \mathbb{Z}$ range over the *integers* and $k \in Key = Key_{\mathrm{P}} \cup Key_{\mathrm{S}}$ range over *keys*, where Key_{P} and Key_{S} are disjoint. The *values* are built up by the *ordinary values*, integers, keys and *pairs* of values, together with the *encrypted values* $u \in U = U_{\mathrm{P}} \cup U_{\mathrm{S}}$.

$$\text{values} \in \mathit{Value} \qquad v ::= n \mid k \mid (v_1, v_2) \mid u$$

The system is parameterized over two *symmetric encryption schemes*—one for each key level γ—represented by triples $S\mathcal{E}_\gamma = (\mathcal{K}_\gamma, \mathcal{E}_\gamma, \mathcal{D}_\gamma)$, where

– $\mathcal{K}_\gamma$ is a *key generation* algorithm that on each invocation generates a new key.
– $\mathcal{E}_\gamma$ is a *probabilistic* encryption algorithm that takes a key $k \in Key_\gamma$, a value $v \in \mathit{Value}$ and returns a ciphertext $u \in U_\gamma$.
– $\mathcal{D}_\gamma$ is a deterministic decryption algorithm that takes a key $k \in Key_\gamma$, a ciphertext $u \in U_\gamma$ and returns a value $v \in \mathit{Value}$ or fails. Decryption should satisfy $\mathcal{D}_\gamma(k, \mathcal{E}_\gamma(k, v)) = v$.

The reason for the use of different encryption schemes for different security levels is to lay the ground for an extension of the system into a *multi-level* system, i.e. a system with more than two security levels. In such a system we would have one encryption schema at each security level, trusted to encrypt values up to and including the security level. We shall assume that the keys sets Key_{P} and Key_{S} of the two different encryption schemes are distinct; let pk range over Key_{P} and sk over Key_{S}.

Input and output is modeled in terms of streams of values with the cons operation "$\cdot$" and the distinguished empty stream ϵ. The *full environment* E consists of four components: (i) the variable environment M, which is a stack of mappings from variable names to lifted values (values joined with a special value for undefined $\mathit{Value}^\bullet = \mathit{Value} \cup \{\bullet\}$); (ii) the key-stream environment G, which maps an encryption scheme level to the *stream of keys* generated by successive use of the key generator (let ks range over streams of keys); (iii) the input environment I and (iv) the output environment O, which map channel names to streams of values.

Semantics of Expressions The evaluation of expressions has the form $\langle (M, G), e\rangle \Downarrow \langle G', v\rangle$: evaluating an expression in a given variable and key-stream environment yields a value and a possibly updated key-stream environment. The semantics of integers, variables, total binary operators, pair formation, and projection are entirely standard.

Figure 3 presents the rules specific to the treatment of cryptography; the rest of the rules can be found in [3]. Key generation (S-NEWKEY) takes the level of the key to be generated and returns the topmost element in the key stream associated to that level in the key-stream environment. Encryption (S-ENC) and decryption (S-DEC) both use the encryption schemes $S\mathcal{E}_\gamma$ introduced above.

Semantics of 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 entirely standard for a while language with channels—everything specific to encryption is in the expressions. For space reasons the semantics of the commands is not presented here but can be found in [3].

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

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

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

Fig. 3. Semantics of Expressions

3 Security

This section states the assumptions our semantic model makes on the underlying encryption schema and shows how these assumptions lead up to a natural formulation of possibilistic noninterference. The section concludes by investigating the relation between our assumptions and common cryptographic attacker models.

Encryption Model As was mentioned above, this paper only considers *probabilistic encryption schemes*. A probabilistic encryption scheme is a triple $(\mathcal{K}, \mathcal{E}, \mathcal{D})$ where the encryption algorithm is a function from a key, a plaintext, and some initial *random data*, referred to as the *initial vector*. Such an algorithm will produce a set of possible ciphertexts for each plaintext-key pair, one ciphertext for each initial vector.

To be able to formulate and prove possibilistic noninterference for our system we need to demand two properties of the underlying encryption schemes. The first property is the assumption that an adversary can learn nothing about the plaintext or the key by observing the ciphertext. This property, known as Shannon's *perfect secrecy* [30], is used to justify our *indistinguishability* relation on ciphertexts.

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:

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

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 public values are equal. However, from the assumption that an adversary can learn nothing about the plaintext from observing the ciphertext it is secure to treat all ciphertexts of the same length[1] as low-equivalent. However appealing this may be, such a treatment leads the ability of masking implicit flows in ciphertexts. Consider the program on Listing 1 for some public channel `ch` and encryption with secret key `k`:

[1] We do not assume that encryption hides the length of messages.

```
l := enc(k, a);
out(ch, l);
if (h) then l := enc(k, b) else skip;
out(ch, l);
```

Listing 1. Occlusion

If all encrypted values are considered equal then we cannot distinguish between the first and the second output value, even though it is clear that the equality/inequality of the first and the second value reflects the secret value h.

Possibilistic Noninterference To address this problem we use a variant of noninterference known as possibilistic noninterference, which allows us to create a notion of low-equivalence that disallows the above example without disallowing intuitively secure uses. Before we formalize our notion of possibilistic noninterference, let us lift the evaluation relation to a set of results as follows:

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

With this we can formulate our notion of possibilistic noninterference. Let $E_1 \sim_\Sigma E_2$ denote that the environments E_1 and E_2 are low-equivalent w.r.t the environment type Σ. A pair of commands, c_1 and c_2 are noninterfering if

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

That is, two commands are considered *equivalent* if, for every pair of low-equivalent environments *in which the commands terminate* it holds that there exists the *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.

By only considering environments for which the commands terminate, we ignore the issue with crashes. This is equivalent to saying that normal and abnormal termination cannot be distinguished by the attacker.

Adequacy of the Model The choice of possibilistic noninterference does not automatically solve the above problem—using the full low-equivalence relation on ciphertexts would lead to the same danger of masking insecure flows. Instead the low-equivalence relation has to be crafted carefully to avoid masked insecure flows and at the same time allow secure usage of encryption primitives. We will now show how this can be done for probabilistic encryption schemes. Consider first what happens in the above example. Let two low-equivalent environments E_1 and E_2 s.t. h is *true* in the first and *false* in the second. The result of running the *if* statement of the example above in the second environment E_2 is the singleton set $\hat{E}_2 = \{E_2\}$. However, the result of running it in the first environment is the set of environments $\hat{E}_1 = \{E_1[l = c] \mid encrypt(b) = c\}$, where each c is obtained by encrypting b under the same key but with different initial vectors. The demand of possibilistic noninterference is that for each environment in $\hat{E}_1$ there should exists a low-equivalent environment in $\hat{E}_2$. This is only the case if all ciphertexts $\{c \mid encrypt(b) = c\}$ are low-equivalent. Thus, any low-equivalence relation that does not consider the different ciphertexts originating from one plaintext and one key to be the equivalent will prevent this kind of masking. However, we must make sure that each ciphertext produced by one plaintext and key has a low-equivalent ciphertext for each other choice of plaintext and key.

Fortunately, for probabilistic encryption schemes we can easily form a low-equivalence relation $\doteq$ with these properties by regarding ciphertexts *with the same random initial vector* to be equivalent:

$$\forall k_1, k_2, v_1, v_2 \; . \; \mathcal{E}(k_1, v_1, iv) \doteq \mathcal{E}(k_2, v_2, iv)$$

where iv ranges over initial vectors. This relation has the following properties: (i) different ciphertexts produced by one plaintext and one key will have different initial vectors and will not be low-equivalent, and (ii) since each plaintext and key will produce ciphertexts using all initial vectors, for each ciphertext produced by one plaintext and key there will be exactly one low-equivalent ciphertext for every other choice of plaintext and key.

Relation to Computational Adversary Models The perfect secrecy and authenticity demands on the encryption schemes are fairly strong. However, there are schemes for which the probability of breaking these assumptions is provably negligible.

The first demand that the ciphertexts should give no information about the plaintexts is commonly relaxed to the notion of *semantic security under chosen plaintext attack* (SEM-CPA) by assuming that the adversary has *limited computational power*. Semantic security states that "*Whatever is efficiently computable about the cleartext given the cyphertext, is also efficiently computable without the cyphertext*" [17]. [2]

In the same way we may allow a relaxation of the demand of authenticity, which can be implemented by combining *Message Authentication Code* (MAC) with a SEM-CPA encryption scheme to form a new scheme that is both secure (SEM-CPA) and authenticity preserving (INT-PTXT)[6]. A scheme is INT-PTXT if the chance that an adversary can produce ciphertexts C s.t. $M = \mathcal{D}_k(C) \neq \bot$ and M was never a parameter of $\mathcal{E}_k(\cdot)$ is negligible. To see that the probability of a successful decryption using the wrong key is negligible under an INT-PTXT scheme consider the following. If a ciphertext $C = \mathcal{E}_k(M)$ decrypts successfully using another key than was used to construct the message i.e. $M' = \mathcal{D}_{k'}(C)$ for $k' \neq k$ then the scheme cannot be INT-PTXT, since M' was never a parameter of $\mathcal{E}_{k'}(\cdot)$.

On Semantic Security We believe that it is possible to prove a general result that if a program with SEM-CPA + INT-PTXT encryption primitives is secure w.r.t. possibilistic noninterference then it is also semantically secure. This result is likely to involve restrictions on *key cycles*, which are a known problem when reconciling the formal and computational views of cryptography [2], or demanding that the underlying schema is secure in the presence of such cycles (cf. *KDM security* [7]).

With such a result at hand, we shall be able 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 semantic security. This opens

[2] There is another frequently used notion of security under a computationally limited adversary, IND-CPA. IND-CPA has been shown to be equivalent to SEM-CPA [17, 6].

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 *secret* or *public*. 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, $\texttt{enc}_{\texttt{S}}\ (\texttt{int H})\ \texttt{L}$ is the type of a secret integer that has been encrypted with a secret key once and $\texttt{enc}_{\texttt{S}}\ (\texttt{enc}_{\texttt{S}}\ (\texttt{int H})\ \texttt{L})\ \texttt{L}$ is the type of an integer that has been encrypted with a secret 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 alike 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, Σ, is the pair of a variable type environment and a channel type environment.

Well-formed Values Well-formedness defines the meaning of the types ignoring the security annotations. The well-formedness is entirely standard and is omitted for space reasons.

Low-equivalence In Figure 4 we formalize the low-equivalence relation. For complex types, i.e., pairs and environments, low-equivalence is defined structurally by demanding the parts of the complex type to be low-equivalent w.r.t. the corresponding type. Any values are low-equivalent w.r.t. a secret type. Integers are low-equivalent w.r.t. a public 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 public key is considered to be of low security and a secret key of high security. Thus, public keys are low-equivalent if they are equal, and any two secret keys are low-equivalent.

The most interesting rule is the rule defining low-equivalence w.r.t. a public encryption type (LE-ENC-L1) and (LE-ENC-L2). These two rules define the difference in meaning between encryption with a secret and a public key. First, in both rules, the encrypted values must be low-equivalent w.r.t. the low-equivalence relation of encrypted values. Second, there must exist a pair of low-equivalent keys w.r.t. the key type of the encryption type that decrypt the encrypted value to two values. This is where the rules differ. Since ciphertexts created by public keys can be decrypted by anyone with access to the public keys, we have to demand that the inside of the encrypted value contains only public values. This is done in the (LE-ENC-L2) rule, which demands that the inside

$$\text{(LE-KEY-L)}\ \frac{}{pk^{\bullet} \sim_{\mathtt{key\ P}} pk^{\bullet}} \qquad \text{(LE-PAIR)}\ \frac{v_{11} \sim_{\tau_1} v_{21} \qquad v_{12} \sim_{\tau_2} v_{22}}{(v_{11}, v_{12}) \sim_{(\tau_1,\tau_2)} (v_{21}, v_{22})}$$

$$\text{(LE-KEY-H)}\ \frac{}{sk_1^{\bullet} \sim_{\mathtt{key\ S}} sk_2^{\bullet}} \qquad \text{(LE-MEM)}\ \frac{\forall x \in \mathtt{dom}\ (\Omega) \qquad M_1(x) \sim_{\Omega(x)} M_2(x)}{M_1 \sim_{\Omega} M_2}$$

$$\text{(LE-INT-L)}\ \frac{}{n^{\bullet} \sim_{\mathtt{int\ L}} n^{\bullet}} \qquad \text{(LE-INENV)}\ \frac{\forall ch \in dom(\Theta)\ .\ I_1(ch) \sim_{\Theta(ch)} I_2(ch)}{I_1 \sim_{\Theta} I_2}$$

$$\text{(LE-INT-H)}\ \frac{}{n_1^{\bullet} \sim_{\mathtt{int\ H}} n_2^{\bullet}}$$

$$\text{(LE-ENC-L3)}\ \frac{}{\bullet \sim_{\mathtt{enc_P}\ \tau\ \mathtt{L}} \bullet} \qquad \text{(LE-OUTENV)}\ \frac{\begin{array}{c}\forall ch \in dom(\Theta)\ . \\ O_1(ch) \sim_{\Theta(ch)} O_2(ch)\end{array}}{O_1 \sim_{\Theta} O_2}$$

$$\text{(LE-ENC-H)}\ \frac{}{u_1^{\bullet} \sim_{\mathtt{enc}_\gamma\ \tau\ \mathtt{H}} u_2^{\bullet}} \qquad \text{(LE-KGEN)}\ \frac{G_1(\mathtt{S}) \sim G_2(\mathtt{S}) \qquad G_1(\mathtt{P}) \sim G_2(\mathtt{P})}{G_1 \sim G_2}$$

$$\text{(LE-KGENP)}\ \frac{\begin{array}{c} pk_1 \sim_{\mathtt{key\ P}} pk_2 \\ K_1 \sim_{\mathtt{P}} K_2 \end{array}}{pk_1 \cdot K_1 \sim_{\mathtt{P}} pk_2 \cdot K_2} \qquad \text{(LE-KGENS)}\ \frac{\begin{array}{c} sk_1 \sim_{\mathtt{key\ S}} sk_2 \\ K_1 \sim_{\mathtt{S}} K_2 \end{array}}{sk_1 \cdot K_1 \sim_{\mathtt{S}} sk_2 \cdot K_2}$$

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

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

Fig. 4. Low-equivalence

is not only low-equivalent w.r.t. its type τ, but low-equivalent w.r.t. $tolow(\tau)$, which is defined as follows:

$$tolow(t\ \sigma) = t\ \mathtt{L} \quad tolow(\mathtt{key\ P}) = \mathtt{key\ P} \quad tolow((\tau_1, \tau_2)) = (tolow(\tau_1), tolow(\tau_2))$$

The (LE-ENC-L1) rule can be seen as encoding the power of the attackers. For encryption with secret keys the demand is only that the resulting values should be low-equivalent w.r.t. the primitive type, τ, of the encryption type. This way, we demand low-equivalence inside encrypted values and make certain that that the result of decrypting low-equivalent encrypted values will result in low-equivalent values and that secret values are not stored inside encrypted values that are created by public keys.

Subtyping The subtyping is entirely standard; it allows public information to be seen as secret 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 subtyping relation can be found in [3].

Expression Type Rules The type rules for expressions are of the form $\Omega, pc \vdash e : \tau$. Figure 5 defines typing rules for non-standard expressions, while the rest of the rules can be found in [3]. 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

$$\text{(T-NEWKEY)}\ \frac{pc \sqsubseteq \mathit{lvl}(\mathtt{key}\ \gamma)}{\Omega, pc \vdash \mathtt{newkey}\ \gamma : \mathtt{key}\ \gamma} \qquad \text{(T-ENC1)}\ \frac{\begin{array}{c}\Omega, pc \vdash e_1 : \mathtt{key\ S} \\ \Omega, pc \vdash e_2 : \tau\end{array}}{\Omega, pc \vdash \mathtt{enc}_{\mathtt{S}}\ (e_1, e_2) : \mathtt{enc}_{\mathtt{S}}\ \tau\ \mathtt{L}}$$

$$\text{(T-ENC2)}\ \frac{\begin{array}{c}\Omega, pc \vdash e_1 : \mathtt{key\ P} \\ \Omega, pc \vdash e_2 : \tau \qquad \mathit{lvl}(\tau) = \sigma\end{array}}{\Omega, pc \vdash \mathtt{enc}_{\mathtt{P}}\ (e_1, e_2) : \mathtt{enc}_{\mathtt{P}}\ \tau\ \sigma} \qquad \text{(T-DEC)}\ \frac{\begin{array}{c}\Omega, pc \vdash e_1 : \mathtt{key}\ \gamma \\ \Omega, pc \vdash e_2 : \mathtt{enc}_{\gamma}\ \tau\ \sigma\end{array}}{\Omega, pc \vdash \mathtt{dec}_{\gamma}\ (e_1, e_2) : \tau^{\sigma}}$$

Fig. 5. Type Rules of Expressions

reason for this is that we assume that the public-key stream is publicly observable. Encryption with secret keys will always result in public encrypted values. Encryption with public keys is possible on any value but produces a result that is as secret as the original value. Both the type rule for key generation and the type rule for public encryption makes use of function $\mathit{lvl}(\cdot)$ that computes the security level of the given value:

$$\mathit{lvl}(t\ \sigma) = \sigma \quad \mathit{lvl}((\tau_1, \tau_2)) = \mathit{lvl}(\tau_1) \sqcup \mathit{lvl}(\tau_2) \quad \mathit{lvl}(\mathtt{key\ P}) = \mathtt{L} \quad \mathit{lvl}(\mathtt{key\ S}) = \mathtt{H}$$

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:

$$(t\ \sigma)^{\sigma'} = t\ (\sigma \sqcup \sigma') \quad (\tau_1, \tau_2)^{\sigma} = (\tau_1^{\sigma}, \tau_2^{\sigma}) \quad (\mathtt{key\ P})^{\mathtt{L}} = \mathtt{key\ P} \quad (\mathtt{key\ S})^{\sigma} = \mathtt{key\ S}$$

Command Type Rules As with expressions most of the rules are standard for a security type system (cf. [34]). As is standard, following Denning's original approach to analyzing programs for secure information flow [13], in order to prevent implicit flows the notion of *security context* is defined. 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 the commands with side-effects, i.e., variable assignment, input, and output. A block of local declarations followed by a sequence of statements is checked by first adding the declared variables to the variable environment and then checking all statements in the new type environment. The type rule for sequences of statements (T-SEQ) checks all statements of the sequence. *If* and *while* are the two constructs that can lead to indirect flows since they affect the control flow. Thus, the body of the *if* and the *while* are checked in the context of the security level of the control expression. This way, when a branch is depending on a secret the body of that branch is prevented from causing any low side effects. The type rules of commands can be found in [3].

5 Soundness

The main soundness theorem of the paper states that well-typed programs are noninterfering. Typically, for typed programming languages, the soundness is phrased in terms of *progress*, i.e. well-typed programs can always be evaluated in well-formed environments, and *preservation*, i.e. after this step has been made the resulting environment is well formed. It may be interesting to note that the way we have avoided to model error makes this system not satisfy progress: decryption with the wrong key or computing with an uninitialized variable will prevent evaluation. The well known solution is to model failure in the semantics. To keep the presentation cleaner we refrain from this.

The soundness theorem states that well-typed programs are noninterfering. Section 3 lifts the evaluation relation of commands to sets and formulates noninterference for commands. Before giving the formulation of the soundness theorem we must lift the codomain of the evaluation relation of expressions to sets and formulate noninterference for expressions:

$$\langle (M,G),e\rangle \Downarrow \langle G',\hat{v}\rangle \text{ iff } \hat{v} = \{v \mid \langle (M,G),e\rangle \Downarrow \langle G',v\rangle\}$$

With this we can 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 key-stream and variable environments, yielding pairs of new key-stream environments and results, then these results should be low-equivalent:

$$\begin{aligned} NI(e_1,e_2)_{\Omega,\tau} \equiv \forall M_1, M_2, G_1, G_2 \,.\, M_1 \sim_\Omega M_2 \wedge G_1 \sim G_2 \wedge \\ \langle (M_i,G_i),e_i\rangle \Downarrow \langle G'_i,\hat{v}_i\rangle \wedge \hat{v}_i \neq \emptyset \Longrightarrow \\ G'_1 \sim G'_2 \wedge \forall v_1 \in \hat{v}_1 \, \exists v_2 \in \hat{v}_2 \,.\, v_1 \sim_\tau v_2 \end{aligned}$$

We arrive at the soundness theorems for expressions and commands, both proved by induction on type derivation [3].

Theorem 1. *Soundness for expressions* $\Omega, pc \vdash e : \tau \Longrightarrow NI(e,e)_{\Omega,\tau}$

Theorem 2. *Soundness for commands* $\Sigma, pc \vdash c \Longrightarrow NI(c,c)_\Sigma$

6 Extensions

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

Integrity Confidentiality classifies information into public and secret, 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—$\texttt{safe}_\mathcal{E}(\alpha, (\sigma, \iota))$ and $\texttt{safe}_\mathcal{D}(\alpha, (\sigma, \iota))$—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

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

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

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

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

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

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

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 choice 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.

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 secret keys would contain the *private* keys and the set of public 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 public, 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 public 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 public channel or not using a secret key for encryption. This section discusses some interesting fragments of these implementations.

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 slightly simplify the syntax with respect to Figure 2 for the sake of readability.

Here, the global declarations contain secret key `K` and low channel `backup`. The type of the latter says that only encrypted high integers may be sent over this channel.

Lines 5 and 7 declare and initialize a high integer variable `data`. Line 6 declares the variable `ctxt` of type `enc secret (int high) low`. On line 8 the value of variable `data` is encrypted with secret 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.

```
 1 global K secret;
 2 backup enc secret (int high) low;
 3
 4 actor Backup {
 5   data int high;
 6   ctxt enc secret (int high) low;
 7   data := ...
 8   ctxt := encrypt(K, data);
 9   out backup ctxt;
10 }
```

Listing 2. Backup code

When recovering data, an actor reads the data from the public 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 high;
3   ctxt enc secret (int high) low;
4   in ctxt backup;
5   data := decrypt(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.

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 consist 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. The full version of this paper [3] contains the implementation for the server S and principal B.)

This program declares two channels: chanS for communicating with the server, 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 secret key of a three-element tuple (expressed by nested pairs). Since the level of the key used for encrypting this tuple is secret, 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.

```
1 global Kas secret;
2 chanS <int low, enc secret
3    (<int low, <int low, key secret>>) low>;
4 chanAB enc secret (int high) low;
5 actor A {
6   idA int low; idB int low; tsA int low;
7   messageToB int high;
8   Kab key secret;
9   //  ... initialization
10  Kab := newkey (secret);
11  out chanS <idA,
12       encrypt(Kas, <tsA,<idB, Kab>>)>;
13  out chanAB encrypt (Kab, messageToB);
14 }
```

Listing 4. WMF Implementation

The new key is generated on line 10. Line 12 constructs the first message of the protocol and sends it to the server. Line 13 uses the newly generated key and sends the secret message to the principal B.

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 [29] 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 masking does not actually model information release. Hence, none of the release dimensions is directly suitable for cryptographically-masked flows.

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 [28].

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 [33, 32] 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 [22], provide computational guarantees for a simple imperative language but with the assumption that keys can be statically distinguished. Mitchell et al. [23, 25] 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 [31]. 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* [12, 15] 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 [14, 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 [26] 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.

9 Conclusions and future work

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 distinguish between ciphertexts. This model has a close connection to probabilistic encryption and, we believe, it naturally connects to computational adversary models (cf. Section 3).

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. 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. We plan to explore a semantic justification of these extensions, crystallizing guarantees provided by the typing rules, and to consider cases studies in which it is critical to achieve these guarantees.

Acknowledgments We wish to thank Martín Abadi and Peeter Laud for helpful comments. This work was supported, in part, by the Swedish Research Council and, in part, by the Information Society Technologies programme 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. M. Abadi and P. Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). *J. of Cryptology*, 15(2):103–127, 2002.
3. A. Askarov, D. Hedin, and A. Sabelfeld. Cryptographically-masked flows. Technical report, Chalmers University of Technology, June 2006. Located at `http://www.cs.chalmers.se/~aaskarov/sas06full.pdf`.
4. 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.
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 C. Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In *Advances in Cryptology - Asiacrypt 2000*, volume 1976 of *LNCS*, pages 531–545, January 2000.
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. 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.
12. 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.
13. D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, July 1977.

14. D. Duggan. Cryptographic types. In *Proc. IEEE Computer Security Foundations Workshop*, pages 238–252, June 2002.
15. 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.
16. J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20, April 1982.
17. S. Goldwasser and S. Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28:270–299, 1984.
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 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.
23. 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.
24. D. McCullough. Noninterference and the composability of security properties. In *Proc. IEEE Symp. on Security and Privacy*, pages 177–186, May 1988.
25. 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.
26. A. C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification and qualified robustness. *J. Computer Security*, 2006. To appear.
27. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, January 2003.
28. 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.
29. A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *Proc. IEEE Computer Security Foundations Workshop*, pages 255–269, June 2005.
30. C. E. Shannon. A mathematical theory of communication. *Bell System Tech. J.*, 27:623–656, 1948.
31. E. Sumii and B. Pierce. Logical relations for encryption. In *Proc. IEEE Computer Security Foundations Workshop*, pages 256–269, June 2001.
32. D. Volpano. Secure introduction of one-way functions. In *Proc. IEEE Computer Security Foundations Workshop*, pages 246–254, July 2000.
33. D. Volpano and G. Smith. Verifying secrets and relative secrecy. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 268–276, January 2000.
34. D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. Computer Security*, 4(3):167–187, 1996.