Computationally sound typing for Non-Interference: The case of deterministic encryption

J. Courant, C. Ene, and Y. Lakhnech

VERIMAG - University Joseph Fourier - CNRS - INPG
2, av. de Vignates, 38610 Gières - FRANCE
{name}@imag.fr

Abstract. Type systems for secure information flow aim to prevent a program from leaking information from variables that hold secret data to variables that hold public data. In this work we present a type system to address *deterministic encryption*. The intuition that encrypting a secret yields a public value, that can be stored in a public variable, is faithful for probabilistic encryption but erroneous for deterministic encryption. We prove the computational soundness of our type system in the concrete security framework.

1 Introduction

The notion of *non-interference* has been introduced in [3], with the aim of capturing unwanted information flow in programs. Non-interference assumes a separation between secret (high, private) variables and public (low) variables and requires that executing the program in two initial states that coincide on the public variables leads to final states that coincide on the public variables. In Dennings' seminal paper [2], an expression is classified H if it contains a secret variable; otherwise, it is classified L. The paper introduces two basic principles to avoid information flow: first, to prevent explicit flow, a H expression may not be assigned to a L variable; second, to prevent implicit flows, an H guarded conditional or loop may not affect L variables. Later, Volpano, Smith, and Irvine [13] casted these principles as a type system and showed that they suffice to ensure non-interference. Since this early work, information flow analysis has been extended to deal with other issues such as nontermination, concurrency, nondeterminism, and exceptions; see [9] for a survey. In many applications, however, it is desirable to allow information to flow from secret to public variables in a controlled way. This is called declassification in the literature. In a useful survey, Sabelfeld & Sands [10] classify declassification techniques according to the following dimensions: "what", "who", "where" and "when".

In this paper, we are interested in *cryptography-based declassification*, where encrypted secret data can be published without leaking information about the secrets. The non-interference setting has been extended in [12] to cope with one-way functions and in [5, 6, 11] to cope with probabilistic encryption. We consider length-preserving deterministic encryption, i.e., block ciphers. These are widely used in practice (DES, AES, Idea, etc.). Non-interference type systems developed for probabilistic encryption are not applicable for deterministic encryption. To illustrate some of the subtleties of deterministic encryption, let us consider the following examples where l, l', l'' are public variables and h, h' are secret variables, νl assigns a value sampled from the uniform distribution to the variable l, $+$ is a bijective operator and $\mathsf{Enc}(k, e)$ denotes the encryption of e with the symmetric key k. We assume that the encryption function $\mathsf{Enc}(k, \cdot)$ is a pseudo-random permutation. A simple program is the following: $l := \mathsf{Enc}(k, h); l' := \mathsf{Enc}(k, h')$. The equality $\mathsf{Enc}(k, h) = \mathsf{Enc}(k, h')$ is almost never true in case of probabilistic encryption, independently whether $h = h'$. Hence, this program does not leak information in case of probabilistic encryption. This is not true in the case of deterministic encryption as we have $h = h'$ if and only if $l = l'$ at program termination. Indeed, deterministic encryption is not repetition concealing in contrast to probabilistic encryption. Consider now, the program $\nu l; l' := \mathsf{Enc}(k, l + h)$, where the value of l is randomly sampled. It does not leak information, even if the attacker is given the value of l. Yet, we have to be careful concerning how the value of l is used. Indeed, the execution of the command $l'' := \mathsf{Enc}(k, l + h')$ at the end of this program would leak information. However, the following slightly modified program : $\nu l; l' := \mathsf{Enc}(k, l + h); l'' := \mathsf{Enc}(k, l' + h')$ does not leak information. Notice that this version corresponds to a simplified block encryption, using the CBC mode: (l, l', l'') can be seen as the cipher text obtained by encrypting the secret (h, h'). Let us consider an example that shows the subtelties that may arise when deterministic encryption is used.

Example 1. In this example (inspired from [11]), '+" is the bitwise-xor operation over blocks of p bits; the other operations are: "$|$" the bitwise-or operation, "$\ll$" the shift-left operation and "$=$" the test for equality. Consider the following command, where h is a private variable and l, m, l_1, l_2 and l_r are public variables.

```
$l := 0^p; m := 0^{p-1}1;$
while_p 1 do    $l_1 := \mathsf{Enc}(k, h|m); l_2 := \mathsf{Enc}(k, h);$
                if $(l_1 = l_2)$ then $l := l|m$ else skip fi ;
                $m := m \ll 1$ od
```

Since encryption is deterministic, this command completely leaks the value of h: it copies h into l. Consider now the following modified command.

```
$l := 0^p; m := 0^{p-1}1;$
while_p 1 do    $\nu l_r; l_1 := \mathsf{Enc}(k, (h|m) + l_r); l_2 := \mathsf{Enc}(k, h + l_r);$
                if $(l_1 = l_2)$ then $l := l|m$ else skip fi ;
                $m := m \ll 1$ od
```

As the same "random l_r" is reused in the second encryption, the obtained code is insecure: it still copies h into l. However, if we re-sample l_r in the second encryption, the command becomes secure.

```
$l := 0^p; m := 0^{p-1}1;$
while_p 1 do    $\nu l_r; l_1 := \mathsf{Enc}(k, (h|m) + l_r); \nu l_r; l_2 := \mathsf{Enc}(k, h + l_r);$
                if $(l_1 = l_2)$ then $l := l|m$ else skip fi ;
                $m := m \ll 1$ od
```

1.1 Contributions

In this paper, we design a type system for information flow for an imperative language that includes block ciphers and show its soundness under the assumption that the encryption scheme is a pseudo-random permutation. Our soundness proof is carried in the concrete (exact) security framework that aims at providing concrete estimates about the security of the considered system.

This is to our knowledge the first time that a type system for non-interference is proven correct in the concrete security framework. One can distinguish three security proof settings: first, the symbolic setting, also called formal and Dolev-Yao, where cryptographic primitives are operators on formal expressions (terms) and security proofs are reachability or observational equivalence proofs; second, the computational setting where cryptographic primitives are algorithms and security proofs are asymptotic based on poly-time reductions; third, the concrete security setting where proofs are also by reduction but no asymptotics are involved and reductions are as efficient as possible.

1.2 Related work

A few works on information flow study computationally sound type systems for non-interference. Peeter Laud has pioneered the area of computationally secure information flow analysis in the presence of encryption. In his first works [4, 5] the analysis was in the form of static analysis and encryption is probabilistic. In more recent work [6] co-authored with Varmo Vene, he presents a type system for information flow in presence of probabilistic encryption. Geoffrey Smith and Rafael Alpízar present in [11] a computationally sound type system for probabilistic encryption. In this work, as in ours, the generation and manipulation of keys is not considered. The main difference, however, to our work is that the above cited works assume probabilistic encryption. Volpano in [12] considers one-way functions. His definition of non-interference is, however, weaker than ours as it essentially means that a well-typed program that leaks information can be used to invert the one-way function. But this does not imply that no information about secret data is learned. Malacaria presents in [7] an information-theoretic definition of non-interference applied to imperative languages with random assignment, and gives an algorithm to approximate the information leaked in a loop. It is easy to prove that for programs that do not use encryption our definition is stronger that his definition. Extending his technique for programs that use encryption does not seem to be immediate.

1.3 Paper structure

In section 2 we introduce some preliminaries including some terminology concerning probabilities, indistinguishability and pseudo-random permutations. In section 3, we present the syntax and semantics for an imperative language build that includes random assignment and deterministic encryption. In section 4 we introduce a type system for randomized expressions, and justify its computational soundness. In section 5, we give a type system for the language presented in section 3 and we prove its computational soundness. The soundness of the type system for this language is proved by two successive reductions: first to a language where the encryption function is interpreted as a random permutation, and then to language where there is no encryption function. Finally, we conclude, and give some possible extensions.

2 Preliminaries

A *finite probability distribution* $\mathcal{D} = (\mathcal{U}, \Pr)$ *over* $\mathcal{U}$ is a finite non-empty set $\mathcal{U}$ equipped with a function $\Pr : \mathcal{U} \to [0,1]$ such that $\sum_{u \in \mathcal{U}} \Pr[u] = 1$. $\mathsf{Distr}(\mathcal{U})$ is the set of distributions on $\mathcal{U}$. The *probability of an event* $A \subseteq \mathcal{U}$ is $\Pr[A] = \sum_{u \in A} \Pr[u]$. A property P over $\mathcal{U}$ can be seen as the event $\{x \in \mathcal{U} \mid P(x)\}$. The *uniform distribution* on $\mathcal{U}$ is such that $\Pr[u] = \frac{1}{|\mathcal{U}|}$, for any $u \in \mathcal{U}$. $[x_1 \overset{r}{\leftarrow} X_1; \ldots x_n \overset{r}{\leftarrow} X_n : e(x_1, \ldots, x_n)]$ denotes the distribution Y such that $\Pr[Y = e] = \sum_{x_1,\ldots,x_n | e(x_1,\ldots,x_n)=e} \Pr[X_1 = x_1] \ldots \Pr[X_n = x_n]$ (thus $[: u]$ is Dirac's point mass δ_u) and $\Pr[x_1 \overset{r}{\leftarrow} X_1; \ldots x_n \overset{r}{\leftarrow} X_n : P(x_1, \ldots, x_n)]$ denotes the probability of the event P over the distribution $[x_1 \overset{r}{\leftarrow} X_1; \ldots x_n \overset{r}{\leftarrow} X_n : (x_1, \ldots, x_n)]$.

Computational indistinguishability Given two distributions $\mathcal{D}$ and $\mathcal{D}'$, and an algorithm $\mathcal{A}$, we define the *advantage* of $\mathcal{A}$ in distinguishing $\mathcal{D}$ and $\mathcal{D}'$ as $\textsc{Adv}(\mathcal{A}, \mathcal{D}, \mathcal{D}') = |\Pr[x \overset{r}{\leftarrow} \mathcal{D} : \mathcal{A}(x) = 1] - \Pr[x \overset{r}{\leftarrow} \mathcal{D}' : \mathcal{A}(x) = 1]|$ (Informally, this advantage quantifies the success of an adversary trying to guess whether some x has been drawn from $\mathcal{D}$ or from $\mathcal{D}'$ and output its guess as a boolean 0/1.) Two distributions $\mathcal{D}$ and $\mathcal{D}'$ are (t, ϵ)-*indistinguishable*, denoted by $\mathcal{D} \sim_{(t,\epsilon)} \mathcal{D}'$, if $\textsc{Adv}(\mathcal{A}, \mathcal{D}, \mathcal{D}') \leq \epsilon$, for any adversary $\mathcal{A}$ running in time bounded by t.

A function f from a set A to the $\mathsf{Distr}(B)$ can be canonically extended to a function $\widehat{f}$ from $\mathsf{Distr}(A)$ to $\mathsf{Distr}(B)$ as follows: $\widehat{f}(X) = [a \overset{r}{\leftarrow} X; b \overset{r}{\leftarrow} f(a) : b]$. We shall tacitly identify $f : A \to \mathsf{Distr}(B)$ with its canonical extension $\widehat{f}$.

A block cipher is a *family of permutations* $\Pi : \mathbf{Keys}(\Pi) \times \mathcal{U} \to \mathcal{U}$, where $\mathbf{Keys}(\Pi)$ is the key space of Π, and for any $k \in Keys(\Pi)$, $\Pi(k, \cdot)$ is a permutation onto $\mathcal{U}$. We use $\mathsf{Enc}(k, \cdot)$ (resp. $\mathsf{Dec}(k, \cdot)$) instead of $\Pi(k, \cdot)$ (resp. $\Pi^{-1}(k, \cdot)$).
Pseudo-randomness. The usual security notion for ciphers (cf.[8]), states that an adversary accessing an oracle $\mathcal{O}_b$ — either $\mathcal{O}_0$, a random permutation, or $\mathcal{O}_1$, the encryption function — has a bounded advantage to guess which one it has been given (or equivalently the value of b). Formally, consider the following experiments parameterized by b, where Perm is the set of all permutations on $\mathcal{U}$:

```
Experiment $\mathbf{PRP}_b(\mathcal{A})$ :
  $k \overset{r}{\leftarrow} \mathbf{Keys}(\Pi)$; $\mathscr{P} \overset{r}{\leftarrow} \mathsf{Perm}$;
  $\mathcal{O}_0 = \mathscr{P}; \mathcal{O}_1 = \mathsf{Enc}(k, \cdot)$;
  $b' \leftarrow \mathcal{A}^{\mathcal{O}_b}()$
```

The $\mathbf{PRP}$ advantage of $\mathcal{A}$ is defined as

$$\textsc{Adv}^{\text{prp}}_{\Pi}(\mathcal{A}) = |\Pr[\mathbf{PRP}_1(\mathcal{A}) = 1] - \Pr[\mathbf{PRP}_0(\mathcal{A}) = 1]|.$$

An encryption scheme Π is a (t, ϵ)-pseudo-random permutation, denoted (t, ϵ)-**PRP**, if for any adversary $\mathcal{A}$ running in time t, $\textsc{Adv}_{\Pi}^{\mathrm{prp}}(\mathcal{A}) \leq \epsilon$.

3 An imperative language with random assignment and deterministic encryption

In this section, we present a simple while-language extended with a random assignment command and deterministic encryption. We then present in following section type systems for its underlying expressions and commands.

3.1 Expressions

We consider a signature with a sort S, a countable set of constant symbols denoted by $n, n_0, n_1, \cdots$ and two binary function symbols $+ : S \times S \to S$ and $g : S \times S \to S$. We restrict the presentation to two function symbols for simplicity. We consider an interpretation for this signature given by a structure $(\mathcal{U}, \mathcal{I}(\cdot), \mathcal{I}_+(\cdot,\cdot), \mathcal{I}_g(\cdot,\cdot))$ such that:

1. $\mathcal{U} = \{0,1\}^p$, where p is an integer. We use $u \overset{r}{\leftarrow} \mathcal{U}$ as an alternative notation for $u \overset{r}{\leftarrow} \mathcal{D}$, where $\mathcal{D}$ is the uniform probability distribution on $\mathcal{U}$.
2. $\mathcal{I}(\cdot)$ is a deterministic algorithm that takes as input a constant symbol n and computes an element $\mathcal{I}(n)$ in $\mathcal{U}$.
3. $\mathcal{I}_+(u, v)$ is the bitwise exclusive or of u and v. (Actually, this can be generalized to any deterministic algorithm such that $[u \overset{r}{\leftarrow} \mathcal{U} : \mathcal{I}_+(u, v)]$ and $[u \overset{r}{\leftarrow} \mathcal{U} : \mathcal{I}_+(v, u)]$ coincide with the uniform distribution on $\mathcal{U}$.)
4. $\mathcal{I}_g(\cdot,\cdot)$ is a deterministic algorithm that given two elements of $\mathcal{U}$, computes an element in $\mathcal{U}$. We denote the function $\lambda(u, v) \cdot \mathcal{I}_g(u, v)$ by $\mathcal{I}(g)$.

The set **Exp** of expressions is given by the following BNF, where metavariable x ranges over a countable set **Var** of identifiers (variables):

$$e ::= x \mid n \mid e_1 + e_2 \mid g(e_1, e_2)$$

A *memory* (or state) is a mapping that associates to each variable a value in $\mathcal{U}$. The set of memories is denoted by Σ. Given a memory σ, we can associate a value $\mathcal{I}(e)\sigma \in \mathcal{U}$ to each expression e in the usual way.

3.2 Commands

The syntax of the **eWhile** language we consider is defined in Figure 1.

$$\begin{aligned} c ::= {} & x := e \mid x := \mathsf{Enc}(k, e) \mid \textbf{skip} \mid \nu x \mid \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \textbf{ fi} \mid \\ & \textbf{while}_n\, e \textbf{ do } c \textbf{ od} \mid c_1 ; c_2 \end{aligned}$$

Fig. 1. Language syntax of **eWhile**

The loop construct is indexed with an integer number n that specifies the maximal number of permitted unfolding of the loop statement. In other words, a loop statement either terminates because the loop condition becomes false or because the limit n is reached. The reason for adding this is that we are only interested in commands whose running time is bounded. The command $x := \mathsf{Enc}(k, e)$ encrypts the value of e with the key k and stores the result in x.

To a command c, we associate as meaning a function from states to distributions on states: $[\![c]\!] : \Sigma \to \mathsf{Distr}(\Sigma)$. The equations defining $[\![c]\!]$ are given in Figure 2. In the sequel, we will assume given a function $\mathbb{T}(c)$ that bounds the running time of the program c.

$$[\![x := e]\!](\sigma) = [: \sigma[\mathcal{I}(e)\sigma/x]] \qquad [\![c_1; c_2]\!] = \widehat{[\![c_2]\!]} \circ [\![c_1]\!]$$
$$[\![\nu x]\!](\sigma) = [u \overset{r}{\leftarrow} \mathcal{U}; \sigma' := \sigma[u/x] : \sigma'] \qquad [\![\mathbf{skip}]\!](\sigma) = [: \sigma]$$
$$[\![x := \mathsf{Enc}(k, e)]\!](\sigma) = [: \sigma[\mathsf{Enc}(k, \mathcal{I}(e)\sigma)/x]]$$
$$[\![\mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{fi}\ ]\!](\sigma) = \mathbf{if}\ (\mathcal{I}(e))\sigma = 1\ \mathbf{then}\ [\![c_1]\!](\sigma)\ \mathbf{else}\ [\![c_2]\!](\sigma)\ \mathbf{fi}$$
$$[\![\mathbf{while}_n\, e\ \mathbf{do}\ c\ \mathbf{od}\ ]\!](\sigma) = \begin{cases} [\![\mathbf{if}\ e\ \mathbf{then}\ c; \mathbf{while}_{n-1}\, e\ \mathbf{do}\ c\ \mathbf{od}\ \mathbf{else}\ \mathbf{skip}\ \mathbf{fi}\ ]\!](\sigma) \\ \quad \text{if } n > 0 \\ [: \sigma];\ \text{otherwise} \end{cases}$$

Fig. 2. Language semantics of **eWhile**

4 Typing expressions

The expressions introduced so far are deterministic in the sense that the value of an expression is determined once σ is fixed. In order to reason about expressions involving random nonces, we introduce *randomized expressions* defined as follows: $re ::= e \mid \nu x \cdot re$. For $\boldsymbol{x} = (x_1, \cdots, x_n)$, we write $\nu \boldsymbol{x} \cdot e$ instead of $\nu x_1 \cdots \nu x_n \cdot e$. Consider a randomized expression re and let σ be a memory. We define $[\![re]\!] : \Sigma \to \mathsf{Distr}(\mathcal{U} \times \Sigma)$ as follows:

1. $[\![e]\!](\sigma) = [: (\mathcal{I}(e)\sigma, \sigma)]$ and

2. $[\![\nu x \cdot re]\!](\sigma) = [u \overset{r}{\leftarrow} \mathcal{U}; \sigma' := \sigma[u/x]; (v, \sigma'') \overset{r}{\leftarrow} [\![re]\!](\sigma') : (v, \sigma'')]$.

Henceforth, let $\mathbb{T}(re)$ be an upper-bound on the time needed to evaluate $[\![re]\!](\sigma)$, for any σ. Given an expression re, let $\mathsf{fvar}(re)$ denote the set of variables that occur free in re, i.e. $\mathsf{fvar}(\nu x_1 \cdots \nu x_n \cdot e) = var(e) \setminus \{x_1, \cdots, x_n\}$. In the following, we write $x \# re$ to mean $x \notin \mathsf{fvar}(re)$, and $x_1, \ldots, x_n \# re_1, \ldots, re_k$ to mean $x_i \# re_j$ for all (i, j) and $x_i \neq x_j$ for all (i, j).

4.1 Typing expressions

The set **TypeExp** of expression types consists of pairs (τ_s, τ_r) with $\tau_s \in \{L, H\}$ and $\tau_r \in \{\top, L^r, H^r\}$. Intuitively, τ_s is the security type; while τ_r is the randomness type. That is, $\top$ means that the expression can be deterministic or randomized; H^r means that it is randomized and contains a ”random seed” that is secret; and L^r means that it is randomized and the ”random seed” might be public. For instance, consider the expression $h_r + l$ with h_r a secret variable whose value is random and l a public variable. Then, it will be typed (H, H^r) as the random seed h_r is secret. On the other hand, $l_r + l$ will be typed (L, L^r) as it does not depend on a secret variable and the random seed is public. Why should we type these expressions differently? The reason is that the expression $(h_r + l) + h$ can be typed public (low) but the expression $(l_r + l) + h$ must be typed secret (high).

A *type environment* maps each variable in **Var** to a security type in $\{L, H\}$. Our type judgements are of the form $\Gamma \vdash e : \tau$, where $e \in$ **Exp** and $\tau \in$ **TypeExp**. We give our typing and sub-typing rules in Figure 3. A few intuition: the sub-typing rule $(H, H^r) \sqsubseteq (L, L^r)$ says that an expression that is randomized with a secret “random seed”, can be downgraded (and in this case, its randomness is made public); the rule $(+)$ takes into account the good properties of $+$, if one of the arguments is randomized (and the random seed is not reused), then their sum is randomized too.

$$L \sqsubseteq H \qquad H^r \sqsubseteq L^r \sqsubseteq \top \qquad \tau \sqsubseteq \tau$$

$$\frac{\tau_s \sqsubseteq \tau_s',\ \tau_r \sqsubseteq \tau_r'}{(\tau_s, \tau_r) \sqsubseteq (\tau_s', \tau_r')} \qquad \frac{\tau_1 \sqsubseteq \tau_2,\ \tau_2 \sqsubseteq \tau_3}{\tau_1 \sqsubseteq \tau_3} \qquad (H, H^r) \sqsubseteq (L, L^r)$$

Subtyping rules

$$\frac{\Gamma(x) = \tau_s}{\Gamma \vdash x : (\tau_s, \top)}\ (\text{var}) \qquad \frac{\Gamma(x) = \tau_s}{\Gamma \vdash \nu x \cdot x : (\tau_s, \tau_s^r)}\ (\text{R-var})$$

$$\frac{-}{\Gamma \vdash n : (L, \top)}\ (\text{int}) \qquad \frac{\Gamma \vdash re : \tau,\ \tau \sqsubseteq \tau'}{\Gamma \vdash re : \tau'}\ (\text{Subt})$$

$$\frac{\begin{array}{c}\Gamma \vdash \nu \boldsymbol{x}_1 \cdot e_1 : (\tau_s, \tau_r)\\ \Gamma \vdash \nu \boldsymbol{x}_2 \cdot e_2 : (\tau_s, \tau_r')\\ \boldsymbol{x}_i \# re_j, \boldsymbol{x}_j,\ \text{for } i \neq j\end{array}}{\Gamma \vdash \nu \boldsymbol{x}_1 \cdot \nu \boldsymbol{x}_2 \cdot (e_1 + e_2) : (\tau_s, \tau_r \sqcap \tau_r')}\ (+) \qquad \frac{\begin{array}{c}\Gamma \vdash \nu \boldsymbol{x}_1 \cdot e_1 : (\tau_s, \top)\\ \Gamma \vdash \nu \boldsymbol{x}_2 \cdot e_2 : (\tau_s, \top)\\ \boldsymbol{x}_i \# re_j, \boldsymbol{x}_j,\ \text{for } i \neq j\end{array}}{\Gamma \vdash \nu \boldsymbol{x}_1 \cdot \nu \boldsymbol{x}_2 \cdot g(e_1, e_2) : (\tau_s, \top)}\ (\text{exp})$$

Typing rules

$$\frac{\Gamma \vdash re : \tau}{\Gamma \vdash \nu x \cdot re : \tau}\ (\nu\text{-Intr}) \qquad \frac{\Gamma \vdash \nu y \cdot \nu x \cdot re : \tau}{\Gamma \vdash \nu x \cdot \nu y \cdot re : \tau}\ (\nu\text{-Comm})$$

Structural rules

Fig. 3. Typing rules for Expressions

Example 2. Let Γ be a type environment such that $\Gamma(h_r) = \Gamma(h) = H$. Then, we have:

$$\dfrac{\dfrac{\dfrac{\Gamma(h_r) = H}{\Gamma \vdash \nu h_r \cdot h_r : (H, H^r)}\ (\text{R-var}) \quad \dfrac{\Gamma(h) = H}{\Gamma \vdash h : (H, \top)}\ (\text{var})}{\Gamma \vdash \nu h_r \cdot (h_r + h) : (H, H^r)}\ (+)}{\Gamma \vdash \nu h_r \cdot (h_r + h) : (L, L^r)}\ (\text{Sixth subtyping rule})$$

Soundness of the type system We now undertake the endeavor to show that expressions typed (L, L^r) do not leak information. In order to rigorously define information leakage, we first introduce *Γ-equivalent distributions.*

Definition 1. *Let X be a distribution on Σ and Γ a type environment. Let $\Gamma^{-1}(L) = \{x \mid \Gamma(x) = L\}$ be the set of low variables and assume that this set is finite. We denote by $\Gamma(X)$ the distribution $[\sigma \overset{r}{\leftarrow} X : \sigma_{|\Gamma^{-1}(L)}]$. Moreover, we write $X =^\Gamma Y$, if $\Gamma(X) = \Gamma(Y)$, and $X \sim^\Gamma_{(t,\epsilon)} Y$, if $\Gamma(X) \sim_{(t,\epsilon)} \Gamma(Y)$. Similarly, for a distribution X on $\mathcal{U} \times \Sigma$, we denote by $\Gamma(X)$ the distribution $[(v, \sigma) \overset{r}{\leftarrow} X : (v, \sigma_{|\Gamma^{-1}(L)})]$.*

The following theorem expresses soundness of our type system for expressions.

Theorem 1. *Let re be an expression, Γ be a type environment and let $X, Y \in \mathsf{Distr}(\Sigma)$ arbitrary distributions.*

- *If $X =^\Gamma Y$ and $\Gamma \vdash re : (L, \top)$, then $[\![re]\!](X) =^\Gamma [\![re]\!](Y)$.*
- *If $X \sim^\Gamma_{(t,\epsilon)} Y$ and $\Gamma \vdash re : (L, \top)$, then $[\![re]\!](X) \sim^\Gamma_{(t - \mathbb{T}(re), \epsilon)} [\![re]\!](Y)$.*

5 A type system for commands

5.1 The typing system

In this section, we present a computationally sound type system for the **eWhile** language of Section 3.2 . We consider programs where applications of Enc have been annotated by r, in case its argument has type (τ, τ^r), and by $\top$, in case it has type $(\tau, \top)$. Recall the following examples from Section 1:

1. $\nu\ell_r; \ell := \mathsf{Enc}^r(k, h + \ell_r); \ell' := \mathsf{Enc}^\top(k, h' + \ell_r)$,
2. $\nu\ell_r; \ell' := \mathsf{Enc}^r(k, h + \ell_r); \ell'' := \mathsf{Enc}^r(k, h' + \ell')$.

The first program is not secure since $h = h'$ iff $\ell = \ell'$. The problem here is that the same random value assigned to ℓ_r is used twice. The second program is secure since the value assigned to ℓ' after the first assignment is indistinguishable from a randomly sampled value. This is due to the properties of the encryption function that we assume to be a pseudo-random permutation. Thus, in order to have a sound type system, we need to forbid the reuse of the same sampled value in two different encryptions; and in order to have a not too restrictive type system, we need to record the variables that are assigned pseudo-random values as a result of the encryption function. This motivates the introduction of the functions $\mathcal{F}$, resp. $\mathcal{G}$, used to compute the propagation of the set of variables that should not be used inside calls of Enc annotated with $\top$, resp. that can be used as random seeds. Informally, variables in the latter set all follow the uniform distribution, and are all independent together and from all variables but the ones in the former set.

$$\begin{array}{ll}
\mathcal{F}(\textbf{skip})(F) & = F \\
\mathcal{F}(\nu x)(F) & = F \setminus \{x\} \\
\mathcal{F}(x := e)(F) & = F \setminus \{x\} \text{ if } \mathsf{fvar}(e) \cap F = \emptyset \\
\mathcal{F}(x := e)(F) & = F \cup \{x\} \text{ otherwise} \\
\mathcal{F}(x := \mathsf{Enc}^r(k, e))(F) & = F \cup \mathsf{fvar}(e) \setminus \{x\} \\
\mathcal{F}(x := \mathsf{Enc}^\top(k, e))(F) & = F \\
\mathcal{F}(c_1; c_2)(F) & = \mathcal{F}(c_2)(\mathcal{F}(c_1)(F)) \\
\mathcal{F}(\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \textbf{ fi })(F) & = \mathcal{F}(c_1)(F) \cup \mathcal{F}(c_2)(F) \\
\mathcal{F}(\textbf{while}_n\, e \textbf{ do } c \textbf{ od })(F) & = \mathcal{F}(c)^\infty(F) \\
\text{where } \mathcal{F}(c)^\infty(F) \text{ is defined as} & \bigcap\{M \mid \mathcal{F}(c)(M) \subseteq M \text{ and } F \subseteq M\}.
\end{array}$$

$$\begin{array}{ll}
\mathcal{G}(\textbf{skip})(G) & = G \\
\mathcal{G}(\nu x)(G) & = G \cup \{x\} \\
\mathcal{G}(x := e)(G) & = G \setminus (\{x\} \cup \mathsf{fvar}(e)) \\
\mathcal{G}(x := \mathsf{Enc}^r(k, e))(G) & = (G \setminus \mathsf{fvar}(e)) \cup \{x\} \\
\mathcal{G}(x := \mathsf{Enc}^\top(k, e))(G) & = G \setminus (\{x\} \cup \mathsf{fvar}(e)) \\
\mathcal{G}(c_1; c_2)(G) & = \mathcal{G}(c_2)(\mathcal{G}(c_1)(G)) \\
\mathcal{G}(\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \textbf{ fi })(G) & = \mathcal{G}(c_1)(G \setminus \mathsf{fvar}(e)) \cap \mathcal{G}(c_2)(G \setminus \mathsf{fvar}(e)) \\
\mathcal{G}(\textbf{while}_n\, e \textbf{ do } c \textbf{ od })(G) & = \mathcal{G}(c)^\infty(G \setminus \mathsf{fvar}(e)) \\
\text{where } \mathcal{G}(c)^\infty(G) \text{ is defined as} & \bigcup\{M \mid M \subseteq \mathcal{G}(c)(M) \text{ and } M \subseteq G\}.
\end{array}$$

Our type judgements have the form $\Gamma, F, G \vdash c : \tau$, where $\tau \in \{L, H\}$ is a security type. The intuitive meaning is the following: in the environment Γ, where the variables in G are assigned random values, and the variables in F are forbidden, c (detectably) affects only variables of type greater than or equal to τ; after its execution, variables in $\mathcal{G}(c)(G)$ have random values, and variables in $\mathcal{F}(c)(F)$ are forbidden. We give the typing and subtyping rules in Figure 4. Our type system ensures that encryption downgrades the security level only in case of random expressions. In other words, $\mathsf{Enc}^\top(k, h)$ has the security level H, and hence, cannot be stored into a low variable, while $\mathsf{Enc}^r(k, h + l_r)$ has security level L, because l_r is a random value that is not used elsewhere. It might appear surprising that the Rule ($\mathsf{Enc}^\top$), which does not allow downgrading, is more restrictive than Rule (Enc^r). To understand this consider the command $\nu\ell_r; \ell := \mathsf{Enc}^r(k, h + \ell_r); \ell' := \mathsf{Enc}^\top(k, \ell_r)$. Leaking the encryption of the low variable ℓ_r allows to check whether $h = 0$, and hence, should be forbidden.

Example 3. This example shows that our system is able to show the security of a cipher block chaining implementation. For simplicity reasons (and because we do not consider arrays yet) we illustrate the case of encrypting two blocks.

$$\frac{}{\Gamma, F, G \vdash \nu x : \Gamma(x)}\ \text{nu-var} \qquad \frac{}{\Gamma, F, G \vdash \textbf{skip} : H}\ \text{skip}$$

$$\frac{\Gamma \vdash \nu G \cdot e : (\Gamma(x), \top)}{\Gamma, F, G \vdash x := e : \Gamma(x)}\ \text{ass} \qquad \frac{\begin{array}{c}\Gamma, F, G \vdash c : \tau \\ G \subseteq G' \quad F' \subseteq F \quad \tau' \sqsubseteq \tau\end{array}}{\Gamma, F', G' \vdash c : \tau'}\ \text{weak}$$

$$\frac{\begin{array}{c}\Gamma \vdash \nu G \cdot e : (\Gamma(x), \top) \\ \mathsf{fvar}(\nu G \cdot e) \cap F = \emptyset\end{array}}{\Gamma, F, G \vdash x := \mathsf{Enc}^{\top}(k, e) : \Gamma(x)}\ \mathsf{Enc}^{\top} \qquad \frac{\Gamma \vdash \nu G \cdot e : (H, L^r)}{\Gamma, F, G \vdash x := \mathsf{Enc}^{r}(k, e) : \Gamma(x)}\ \mathsf{Enc}^{r}$$

$$\frac{\begin{array}{c}\Gamma, F, G \setminus \mathsf{fvar}(e) \vdash c_1 : \tau \\ \Gamma, F, G \setminus \mathsf{fvar}(e) \vdash c_2 : \tau \\ \Gamma \vdash \nu G \cdot e : (\tau, \top) \\ \mathsf{fvar}(\nu G \cdot e) \cap F = \emptyset\end{array}}{\Gamma, F, G \vdash \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \textbf{ fi } : \tau}\ \text{if} \qquad \frac{\begin{array}{c}\Gamma, \mathcal{F}(c)^{\infty}(F), \mathcal{G}(c)^{\infty}(G) \vdash c : \tau \\ \Gamma \vdash \nu \mathcal{G}(c)^{\infty}(G) \cdot e : (\tau, \top) \\ \mathsf{fvar}(\nu \mathcal{G}(c)^{\infty}(G) \cdot e) \cap \mathcal{F}(c)^{\infty}(F) = \emptyset\end{array}}{\Gamma, F, G \vdash \textbf{while}_n\, e \textbf{ do } c \textbf{ od } : \tau}\ \text{while}$$

$$\frac{\Gamma, F_1, G_1 \vdash c_1 : \tau \quad \Gamma, \mathcal{F}(c_1)(F_1), \mathcal{G}(c_1)(G_1) \vdash c_2 : \tau}{\Gamma, F_1, G_1 \vdash c_1; c_2 : \tau}\ \text{seq}$$

Fig. 4. Type systems for commands in **eWhile**

$\nu l_0;$
$l_1 := \mathsf{Enc}(k, l_0 + h_1);$
$l_2 := \mathsf{Enc}(k, l_1 + h_2);$

Let Γ be a type environment such that $\Gamma(h_0) = \Gamma(h_1) = H$ and $\Gamma(l_0) = \Gamma(l_1) = \Gamma(l_2) = L$. This program can be typed in our system as follows:

$$\frac{\Gamma, \emptyset, \emptyset \vdash \nu l_0 : L \qquad \dfrac{\dfrac{\dfrac{\dfrac{\Gamma(l_0) = L}{\Gamma \vdash \nu l_0 \cdot l_0 : (L, L^r)}\ \text{(R-var)}}{\Gamma \vdash \nu l_0 \cdot (l_0 + h_1) : (H, L^r)}\ (+)}{\Gamma, \emptyset, \{l_0\} \vdash l_1 := \mathsf{Enc}(k, l_0 + h_1) : L}\ \text{(ass)} \quad \dfrac{\dfrac{\dfrac{\Gamma(l_1) = L}{\Gamma \vdash \nu l_1 \cdot l_1 : (L, L^r)}\ \text{(R-var)}}{\Gamma \vdash \nu l_1 \cdot (l_1 + h_2) : (H, L^r)}\ (+)}{\Gamma, \{l_0, h_1\}, \{l_1\} \vdash l_2 := \mathsf{Enc}(k, l_1 + h_2) : L}\ \text{(ass)}}{\Gamma, \emptyset, \{l_0\} \vdash l_1 := \mathsf{Enc}(k, l_0 + h_1); l_2 := \mathsf{Enc}(k, l_1 + h_2) : L}\ \text{(seq)}}{\Gamma, \emptyset, \emptyset \vdash \nu l_0; l_1 := \mathsf{Enc}(k, l_0 + h_1); l_2 := \mathsf{Enc}(k, l_1 + h_2) : L}\ \text{(seq)}$$

5.2 Soundness of the typing system of eWhile

In this section, we state the soundness of the type system of the **eWhile** language and sketch its proof. The detailed proof is given in [1].

Let $\mathcal{T}(c)$ denote an upper bound on the number of Enc^r and $\mathsf{Enc}^{\top}$ calls that can be executed during any run of c. Notice that because the running time of c is bounded such a bound exists. Then, we can state the following theorem:

Theorem 2. *Let c be a program, let Γ be a type environment and let Π be an encryption scheme. Moreover, let X and Y be two distributions.*

*If Π is (t', ϵ')-**PRP**, $X \sim^{\Gamma}_{(t,\epsilon)} Y$ and $\Gamma, \emptyset, \emptyset \vdash c : \tau$ then $[\![c]\!](X) \sim^{\Gamma}_{(t'',\epsilon'')} [\![c]\!](Y)$ with $t'' = \min(t - \mathbb{T}(c), t' - \mathbb{T}(c))$ and $\epsilon'' = \epsilon + 2\epsilon' + \frac{2\mathcal{T}(c)^2}{|\mathcal{U}|}$.*

Proof. (*Sketch*). Let **rWhile** denote the set of programs without any call to $\mathsf{Enc}(k, \cdot)$ and **pWhile** denote the set of programs where the encryption function

$\mathsf{Enc}(k,\cdot)$ is interpreted as a random permutation. The main idea of the soundness proof is as follows. Consider a command c with $\Gamma, \emptyset, \emptyset \vdash c : \tau$. Then, let $[\![c]\!]^{\pi}$ denote its interpretation in **pWhile** and let c^r obtained from c by replacing $x := \mathsf{Enc}^r(k,e)$ by νx and $x := \mathsf{Enc}^{\top}(k,e)$ by $g(e)$. Then, we can prove the following statements:

Proposition 1. *For any distribution Z, we have*

1. $[\![c]\!]^{\pi}(Z) \sim_{(t'-\mathbb{T}(c),\epsilon')} [\![c]\!](Z)$ *and*
2. $[\![c]\!]^{\pi}(Z)$ *and* $[\![c^r]\!](Z)$ *are* $\frac{\mathcal{T}(c)^2}{|\mathcal{U}|}$*-statistically close.*

We can also prove the following soundness result of our type system for **rWhile**:

Proposition 2. *Let c be a command in **rWhile**, let Γ be a type environment and let $X, Y \in \mathsf{Distr}(\Sigma)$ be arbitrary distributions.*
If $X \sim^{\Gamma}_{(t,\epsilon)} Y$ and $\Gamma, \emptyset, \emptyset \vdash c : \tau$ then $[\![c]\!](X) \sim^{\Gamma}_{(t-\mathbb{T}(c),\epsilon)} [\![c]\!](Y)$.

From Propositions 1 and 2, we obtain the theorem by transitivity.

Proof sketch of Proposition 1 Let us consider the first item. Let $\mathcal{A}$ be an adversary trying to distinguish $[\![c]\!](Z)$ and $[\![c]\!]^{\pi}(Z)$. We construct an adversary $\mathcal{B}$ against the encryption scheme Π, that runs in time $t' + \mathbb{T}(c)$ and whose advantage is the same as $\mathcal{A}$'s advantage. The adversary $\mathcal{B}$ runs an experiment for $\mathcal{A}$ against $[\![c]\!](Z)$ and $[\![c]\!]^{\pi}(Z)$ using his oracles. First, $\mathcal{B}$ executes the command c using its encryption oracle. That is, whenever a command $x := \mathsf{Enc}(k,e)$ is to be executed in the command c, $\mathcal{B}$ computes the value of e and calls its encryption oracle. After termination of the command c in some state σ, $\mathcal{B}$ runs $\mathcal{A}$ on σ and gives the same answer as $\mathcal{A}$. Formally:

Adversary $\mathcal{B}^{\mathcal{O}_b}$

$$b \xleftarrow{r} \{0,1\}; \quad \sigma \xleftarrow{r} [\![c]\!]^{\mathcal{O}_b}(Z); \quad \mathcal{A}(\sigma).$$

Now it is clear that $\textsc{Adv}^{\mathrm{prp}}_{\Pi}(\mathcal{B}) = \textsc{Adv}(\mathcal{A}, [\![c]\!](Z), [\![c]\!]^{\pi}(Z))$. Moreover, the running time of $\mathcal{B}$ is $\mathcal{A}$'s running time augmented with the time need for computing $[\![c]\!](Z)$, i.e. $\mathbb{T}(c)$. We conclude that $[\![c]\!]^{\pi}(Z) \sim_{(t'-\mathbb{T}(c),\epsilon')} [\![c]\!](Z)$.

Consider now the second item. Roughly speaking, the bound $\frac{\mathcal{T}(c)^2}{|\mathcal{U}|}$ corresponds to the probability of collisions between arguments of Enc^r among themselves and with with arguments of $\mathsf{Enc}^{\top}$; and collisions among values returned by ν. Moreover, we can then prove that c^r is a well-typed **rWhile** program. □

6 Conclusion

This extended abstract introduces a type system for an imperative language that includes deterministic encryption and random assignment. It establishes soundness of the type system under the assumption that the encryption scheme is a pseudo-random permutation. The proof is carried in the concrete security setting, thus providing concrete security estimates. Our work can be extended in several directions. First, we could consider encryption as "first class" expressions.

This is not a substantial extension as any such program can be easily translated into our language and refining the type of variables to (τ_s, τ_r) as for expressions. Second, we could consider decryption. An easy way to do this is to type the result of any decryption with H. This may not, however, be satisfactory as the so-obtained type system would be too restrictive. An other extension consists in considering generation and manipulation of keys - it is not difficult to extend the type system to deal with this, we need, however, to introduce conditions on the expressions (acyclicity) and to apply hybrid arguments; data integrity - which are in some sense dual to non-interference. Some of these extensions are considered in the full paper [1], which also contains the detailed proofs of the results presented in this extended abstract. In the full paper, we also show that our notion of non-interference implies semantic security and Laud's notion.

References

[1] J. Courant, C. Ene, and Y. Lakhnech. Computationally sound typing for non-interference: The case of deterministic encryption. Technical report, VERIMAG- University of Grenoble and CNRS, 2007.

[2] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977.

[3] Joseph A. Goguen and José Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[4] Peeter Laud. Semantics and program analysis of computationally secure information flow. In *ESOP*, pages 77–91, 2001.

[5] Peeter Laud. Handling encryption in an analysis for secure information flow. In *ESOP*, pages 159–173, 2003.

[6] Peeter Laud and Varmo Vene. A type system for computationally secure information flow. In Maciej Liskiewicz and Rüdiger Reischuk, editors, *FCT*, volume 3623 of *LNCS*, pages 365–377. Springer, 2005.

[7] Pasquale Malacaria. Assessing security threats of looping constructs. In Martin Hofmann and Matthias Felleisen, editors, *POPL*. ACM, 2007.

[8] Duong Hieu Phan and David Pointcheval. About the security of ciphers (semantic security and pseudo-random permutations). In Helena Handschuh and M. Anwar Hasan, editors, *Selected Areas in Cryptography*, volume 3357 of *LNCS*, pages 182–197. Springer, 2004.

[9] A. Sabelfeld and A. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Comunications*, 21:5–19, January 2003.

[10] Andrei Sabelfeld and David Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, 2007.

[11] Geoffrey Smith and Rafael Alpzar. Secure information flow with random assignment and encryption. In *FMSE*, pages 33–44, 2006.

[12] Dennis M. Volpano. Secure introduction of one-way functions. In *CSFW*, pages 246–254, 2000.

[13] Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2/3):167–188, 1996.