On the Computational Soundness of Cryptographically Masked Flows

Peeter Laud

Tartu University, Institute of Computer Science, and Cybernetica AS

peeter.laud@ut.ee

Abstract

To speak about the security of information flow in programs employing cryptographic operations, definitions based on computational indistinguishability of distributions over program states have to be used. These definitions, as well as the accompanying analysis tools, are complex and error-prone to argue about. *Cryptographically masked flows*, proposed by Askarov, Hedin and Sabelfeld, are an abstract execution model and security definition that attempt to abstract away the details of computational security. This abstract model is useful because analysis of programs can be conducted using the usual techniques for enforcing non-interference.

In this paper we investigate under which conditions this abstract model is computationally sound, i.e. when does the security of a program in their model imply the computational security of this program. This paper spells out a reasonable set of conditions and then proposes a simpler abstract model that is nevertheless no more restrictive than the cryptographically masked flows together with these conditions for soundness.

Categories and Subject Descriptors F.3.2 [*Semantics of Programming Languages*]: Operational Semantics, Program Analysis

General Terms Languages, Security, Verification

Keywords Secure Information Flow, Encryption, Computational Soundness, Cryptographically Masked Flows

1. Introduction

Non-interference (Goguen and Meseguer 1982) is the usual way of defining the secure information flow in programs (Sabelfeld and Myers 2003). It states that varying only the secret inputs of the program must not change the public outputs — public outputs are determined by the non-secret inputs only. For programs containing encryption or other cryptographic operations, such a definition may be too strong, because a ciphertext still depends on the plaintext used to produce it, albeit in a manner that cannot be exploited by an adversary that uses only a reasonable amount of resources. Instead, the notion of computational non-interference (Laud 2003) has to be used. The definition of computational non-interference is quite complex in its structure and in the used domains. The usage of such a definition requires the semantics of the program to be probabilistic, making it more difficult to argue about. Also, the precise program analyses based directly on computational non-interference (Laud 2001, 2003; Laud and Vene 2005) tend to have a complex structure and their correctness is not always so obvious.

It would be nice to have a more "abstract" definition for the semantics of the programming language, as well as for the security of the information flow, such that

(i) the used domains and the structure of definitions are more conventional;

(ii) the security of the program in the abstract model would imply its security in the computational model;

(iii) the abstraction would hide the cryptographic details of the encryption (including the necessary use of probabilistic domains), but not much else of the computational model.

Cryptographically masked flows by Askarov et al. (2006) aims to be such a more abstract model. This model considers an imperative programming language (in the original paper this language is quite feature-rich; we consider a stripped-down version of it) with key generation, encryption and decryption as distinguished operations. In the concrete semantics, corresponding to the real-world implementations of the language, the encryption operation is probabilistic (otherwise it cannot be sufficiently secure). The semantics of a program maps the initial state S^{i} to a probability distribution D^{f} over final states. In the model of cryptographically masked flows, henceforth called the *abstract semantics*, the encryption operation is non-deterministic — the encryption algorithm works in the same way as in the concrete semantics, but each time it flips a coin it gets both 0 and 1 as the result. The abstract semantics of a program maps the initial state S^{i} to a set $\mathbf{S}^{\mathrm{f}}$ of possible final states. The set $\mathbf{S}^{\mathrm{f}}$ can be obtained from the distribution D^{f} by just forgetting the probabilities (at least if there are no other probabilistic operations except key generation and encryption; such requirement is put forth by Askarov et al. (2006)). The definition of secure information flow in the setting of cryptographically masked flows is the conventional possibilistic non-interference (Smith and Volpano 1998) stating that the set of the low-slices of possible final states may not depend on the initial secrets. However, when considering whether the low-slices of two states are equal, we sometimes allow the values of the variables containing ciphertexts to differ. The equivalence of low-slices is defined so that generally all ciphertexts are considered equal, but we can distinguish a pair of two different ciphertexts from a pair of equal ciphertexts. Cryptographically masked flows can hence be said to satisfy the objectives (i) and (iii). The aim of the current paper is to investigate, to what extent and under which conditions the objective (ii) is satisfied.

Askarov et al. (2006) also give a type system for checking whether a program satisfies the non-interference property given by the cryptographically masked flows. In the current paper, we do

POPL'08, January 7–12, 2008, San Francisco, California, USA.

not treat this type system in any way; we are interested strictly only in the cryptographically masked flows. Still, the results of this paper and Askarov et al. (2006) together establish that if a program is typable according to this type system, and also satisfies certain conditions that we put on it in this paper, then this program has computationally secure information flow.

In the following we will give a precise definition of the programming language and its concrete and abstract semantics in Sec. 2. We continue by formally stating the definitions of secure information flow in both the concrete and abstract settings in Sec. 3. In Sec. 4 we state and discuss the security definitions that the encryption systems employed in this paper must satisfy. In Sec. 5 we give several examples of programs that seem to violate the computational soundness of cryptographically masked flows and outline the conditions that would exclude such programs, these conditions are formally stated in Sec. 6 and their sufficiency is proved in Sec. 7. Reflecting on the constraints put on the programs we devise a new model for abstract execution and non-interference that is simpler and less restrictive; we describe it in Sec. 8. We finish with a review of some related work in Sec. 9 and discussion on desired properties of such abstractions in general in Sec. 10.

2. Programming language

In this paper we consider programs P in the usual WHILE-language defined by

$$P ::= x := o(x_1, \ldots, x_k) \mid \mathit{skip} \mid P_1 \,;\, P_2$$
$$\mid \mathit{if}\ b\ \mathit{then}\ P_1\ \mathit{else}\ P_2 \mid \mathit{while}\ b\ \mathit{do}\ P'$$

where $b, x, x_1, \ldots, x_k$ are variables from a given set $\mathbf{Var}$ and o ranges over a fixed set of arithmetic, relational, boolean, etc. operations. Among the operations of the language we will handle the following ones in a special way: new key generation newkey, (symmetric) encryption enc, decryption dec, pairing $(,)$ and projections π_1 and π_2. The language in (Askarov et al. 2006) is much richer, but the subset we are considering here represents the underlying problem well. As usually done in this area (research on secure information flow), we consider only terminating programs — the issue of leaking information by non-termination (or by execution time) is orthogonal to the issues considered here and can be mitigated by known methods (Agat 2000).

Askarov et al. (2006) give a big-step operational semantics for the programming language. The semantics is quite typeful — values from different sources have different types (key, ciphertext, pair, integer) and if the arguments of the operations are not of the right type, the operation gets stuck (in some sense, by disallowing type errors, the semantics already contains some aspects of the enforcement of non-interference). In this paper, we have tried to simplify the operational semantics as much as possible, leaving out such constraints. Rushing ahead, those constraints actually turn out to be necessary for the soundness result, thus they'll appear again in Sec. 6.

The abstract semantics of expressions is given in Fig. 1. The semantics of an n-ary "normal" operation o is a polynomial-time computable function $[\![o]\!] : \mathbf{Val}^n \to \mathbf{Val}$ where $\mathbf{Val} = \{0,1\}^*$ is the set of values (no operations are probabilistic, except key generation and encryption). We assume that there is a distinguished value $\bot \in \mathbf{Val}$ denoting failure, and that all operations are strict with respect to $\bot$. For giving semantics to pairing and projections, we assume that an easily computable and reversible injective function $\rho : \mathbf{Val}^2 \to \mathbf{Val}$ is fixed; this function is the semantics of the pairing operation. For giving semantics to the key generation, encryption and decryption operations, we fix an encryption system $(\mathcal{K}, \mathcal{E}, \mathcal{D})$. Here $\mathcal{K}$ is the key generation algorithm, $\mathcal{E}$ is the encryption and $\mathcal{D}$ the decryption algorithm. The algorithms $\mathcal{K}$ and $\mathcal{E}$ are

$$\frac{M(x) = v}{\langle M, x\rangle \Downarrow^{\mathrm{a}} v}$$

$$\frac{\langle M, e_i\rangle \Downarrow^{\mathrm{a}} v_i \quad [\![o]\!](v_1, \ldots, v_k) = v}{\langle M, o(e_1, \ldots, e_k)\rangle \Downarrow^{\mathrm{a}} v}$$

$$\frac{\langle M, e\rangle \Downarrow^{\mathrm{a}} u \quad \rho(v_1, v_2) = u}{\langle M, \pi_i(e)\rangle \Downarrow^{\mathrm{a}} v_i}$$

$$\frac{\langle M, e\rangle \Downarrow^{\mathrm{a}} u \quad u \notin \operatorname{Dom} \rho}{\langle M, \pi_i(e)\rangle \Downarrow^{\mathrm{a}} \bot}$$

$$\frac{v \in \operatorname{Supp} \mathcal{K}()}{\langle M, \mathsf{newkey}\rangle \Downarrow^{\mathrm{a}} v}$$

$$\frac{\langle M, e_k\rangle \Downarrow^{\mathrm{a}} v_k \quad \langle M, e_x\rangle \Downarrow^{\mathrm{a}} v_x \quad v_y \in \operatorname{Supp} \mathcal{E}(v_k, v_x)}{\langle M, \mathsf{enc}(e_k, e_x)\rangle \Downarrow^{\mathrm{a}} v_y}$$

Figure 1. Abstract semantics of expressions

probabilistic, $\mathcal{D}$ is deterministic. The algorithm $\mathcal{K}$ takes no arguments, the algorithms $\mathcal{E}$ and $\mathcal{D}$ take two — the key and the plain-/ciphertext. For all keys k that can be output by $\mathcal{K}$, for all plaintexts $x \in \mathbf{Val}$ and all ciphertexts y that can be output by $\mathcal{E}(k, x)$, the equality $\mathcal{D}(k, y) = x$ must hold. The decryption is allowed to fail, it has to produce $\bot$ then. The semantics of the operation dec is $\mathcal{D}$. For a probability distribution D we denote by $\operatorname{Supp} D$ the set of all such x where $D(x) > 0$.

The ability of operations, particularly decryption, to fail is different from (Askarov et al. 2006). In their treatment, the failures are invisible — the executions where an operation is about to fail just get stuck and do not contribute anything to the final set of states. In our opinion this is unrealistic, as the failures are definitely visible in the real world.

In Fig. 1, M is a *memory* — a mapping from variables to values. We see that $\Downarrow^{\mathrm{a}}$ is non-deterministic — the key generations and encryptions may have several possible values. Compared to the treatment of Askarov et al. (2006), we have made a simplification — they let the program state also to contain the stream of yet-to-be-generated keys and the operation newkey takes and returns the first element of that stream (this detail turns out to be important in the security definition, we will discuss it in Sec. 5).

Having defined the abstract semantics of expressions $\Downarrow^{\mathrm{a}}$ we now define the transition relation of the abstract small-step structural operational semantics $\xrightarrow{\mathrm{a}}$ of the programming language. The relation $\xrightarrow{\mathrm{a}}$ relates program configurations $\langle M, P\rangle$ to program configurations or memories. The definition of $\xrightarrow{\mathrm{a}}$ is completely standard (Nielson and Nielson 1992, Chap. 2.2) and is omitted. But note that the non-determinism of $\Downarrow^{\mathrm{a}}$ also causes the non-determinism of $\xrightarrow{\mathrm{a}}$. We also define $\langle M, P\rangle \stackrel{\mathrm{a}}{\Longrightarrow} \mathcal{M}$ if $\mathcal{M} = \{M' \mid \langle M, P\rangle \xrightarrow{\mathrm{a}}{}^{*} M'\}$.

On the concrete / computational side, the non-deterministic constructs are replaced by probabilistic ones. We define

- $\Downarrow^{\mathrm{c}}$, relating an expression e to the probability distribution of its values;
- the transition relation $\xrightarrow{\mathrm{c}}$ of the concrete small-step structural operational semantics, relating a program configuration $\langle M, P\rangle$ to a probability distribution D over memories or to a pair $\langle D, P'\rangle$.

(these straightforward definitions are omitted) Given $\xrightarrow{\mathrm{c}}$ we can define the probability of a sequence $\langle M_0, P_0\rangle \to \langle M_1, P_1\rangle \to \cdots \to \langle M_{n-1}, P_{n-1}\rangle \to M_n$ as the product of the probabilities

$p_1, \ldots, p_n$ where p_i is the probability of M_i in the distribution D_i and D_i is given by $\langle M_{i-1}, P_{i-1}\rangle \xrightarrow{c} \langle D_i, P_i\rangle$ (for $1 \le i \le n-1$) or $\langle M_{n-1}, P_{n-1}\rangle \xrightarrow{c} D_n$. Finally we define $\langle M, P\rangle \stackrel{c}{\Longrightarrow} D$ relating the initial program memory M and the program P to the probability distribution D over final program memories. Here the probability assigned by D to some memory M' is the sum of probabilities of all sequences $\langle M, P\rangle \to \cdots \to M'$. The mapping D is indeed a probability distribution (i.e. all probabilities assigned by it sum up to 1) because we have disallowed the non-termination of P.

3. Security definition

Let $\mathbf{Var}_\mathrm{S}, \mathbf{Var}_\mathrm{P} \subseteq \mathbf{Var}$ be the sets of initial secret and final public variables — we want the adversary that learns the final values of the variables in $\mathbf{Var}_\mathrm{P}$ to be unable to deduce anything it did not know before about the initial values of the variables in $\mathbf{Var}_\mathrm{S}$. We demand $\mathbf{Var}_\mathrm{S} \cap \mathbf{Var}_\mathrm{P} = \emptyset$, but not $\mathbf{Var}_\mathrm{S} \cup \mathbf{Var}_\mathrm{P} = \mathbf{Var}$ — there may be auxiliary variables that do belong to neither $\mathbf{Var}_\mathrm{S}$ nor $\mathbf{Var}_\mathrm{P}$. We demand that the program does not use the initial values of such auxiliary variables.

To give the definition of non-interference in the abstract setting, Askarov et al. (2006) first defined when two cryptotexts "look the same". For bit-strings y_1, y_2 they defined $y_1 =_\mathrm{E} y_2$ if there exist such r, k_1, k_2, x_1, x_2, such that $y_1 = \mathcal{E}(r; k_1, x_1)$ and $y_2 = \mathcal{E}(r; k_2, x_2)$. Here the argument r denotes the choice of the random coins for the algorithm $\mathcal{E}$. I.e. $y_1 =_\mathrm{E} y_2$ if they could be ciphertexts generated with the same random coins (initial vectors).

This relaxed equality is applied only to ciphertexts. To use it for the values of program variables we have to fix which of those variables contain ciphertexts. Hence let $\mathbf{Var}_\mathrm{E} \subseteq \mathbf{Var}_\mathrm{P}$ be the public variables that are assumed to contain ciphertexts. For program memories M_1 and M_2 we define the *low-equivalence* of M_1 and M_2, denoted $M_1 \sim_\mathrm{P} M_2$, if $M_1(x) = M_2(x)$ for all $x \in \mathbf{Var}_\mathrm{P} \backslash \mathbf{Var}_\mathrm{E}$ and $M_1(x) =_\mathrm{E} M_2(x)$ for all $x \in \mathbf{Var}_\mathrm{E}$.

Askarov et al. (2006) define a program P to be non-interfering if for all program memories M_1, M_2, such that $M_1 \sim_\mathrm{P} M_2$ and for all memories M_1', such that $\langle M_1, P\rangle \xrightarrow{a}{}^* M_1'$ there exists a memory M_2', such that $\langle M_2, P\rangle \xrightarrow{a}{}^* M_2'$ and $M_1' \sim_\mathrm{P} M_2'$. This definition gives us the standard nondeterministic non-interference, only the definition of equality for values has been changed.

In the computational setting, the non-interference is defined as the computational independence of secret inputs and public outputs. To define the asymptotic computational notions we need a security parameter n relative to which the asymptotics are taken. Hence let the semantics of the operations be parametrized by this security parameter and let their running time be polynomial with respect to this parameter. In particular, the algorithms $\mathcal{K}$, $\mathcal{E}$ and $\mathcal{D}$ take the security parameter as an extra argument. The semantic relations $\Downarrow^\mathrm{c}$, $\xrightarrow{c}$ and $\stackrel{c}{\Longrightarrow}$ are thus parametrized with n, too.

Let D be the probability distribution of initial memories for the program P; the adversary knows this distribution. Actually, because of the security parameter, $D = \{D_n\}_{n\in\mathbb{N}}$ is a family of probability distributions over program memories. The program P is computationally non-interferent with respect to D (Laud 2003) if the families of probability distributions (parametrized by n)

$$\{\!|(M|_{\mathbf{Var}_\mathrm{S}}, T|_{\mathbf{Var}_\mathrm{P}}) \mid M \leftarrow D_n, \langle M, P\rangle \stackrel{c}{\Longrightarrow}_n \hat{D}, T \leftarrow \hat{D}|\!\}$$

and

$$\{\!|(M'|_{\mathbf{Var}_\mathrm{S}}, T|_{\mathbf{Var}_\mathrm{P}}) \mid M, M' \leftarrow D_n, \langle M, P\rangle \stackrel{c}{\Longrightarrow}_n \hat{D}, T \leftarrow \hat{D}|\!\}$$

are computationally indistinguishable. Here $\{\!|E \mid C|\!\}$ denotes the distribution of the random expression E under the conditions C. The notation $M \leftarrow D_n$ means that the random variable is distributed according to D_n. Hence the first of the above distributions is that of the initial values of the secret variables and final values of the public variables, where the initial memory M is sampled according to D_n and the final memory T is obtained by executing the program P (which is probabilistic) on M. The second of the above distributions is that of the same values of the same variables, but here the initial memory M' and the final memory T correspond to different runs (the initial memories M and M' are sampled independently of each other). Two families of probability distributions $D = \{D_n\}_{n\in\mathbb{N}}$ and $D' = \{D_n'\}_{n\in\mathbb{N}}$ are *computationally indistinguishable* (this is the cryptographic equivalent for being "the same") if for all probabilistic polynomial-time (PPT) adversaries $\mathcal{A}$ the difference

$$\Pr[\mathcal{A}(n, x) = 1 \mid x \leftarrow D_n] - \Pr[\mathcal{A}(n, x) = 1 \mid x \leftarrow D_n']$$

is negligible in n. A function f is negligible if f is $o(1/p)$ for any polynomial p.

Our desired result We want to have a soundness theorem with more or less the following wording:

> Let P be a program. If P satisfies certain conditions and the initial probability distribution D satisfies certain conditions and P is non-interferent in the abstract setting then P is computationally non-interferent with respect to the initial distribution D.

Here the conditions on D should be something natural, for example the independence of the secret values from the public ones. The conditions on P should be verifiable in the abstract setting, otherwise we lose the modularity of the approach. In the Sec. 5 we take a look at what these conditions could be. Let us call a program P *well-structured* if it obeys those conditions.

In the following we often have to speak about the public part of the result of some computation. For a memory M we thus define M_P as the restriction of M to $\mathbf{Var}_\mathrm{P}$. For a set of memories $\mathcal{M}$ we define $\mathcal{M}_\mathrm{P} = \{M_\mathrm{P} \mid M \in \mathcal{M}\}$. For a probability distribution D over memories we have to collapse all memories with the same public part — we define $D_\mathrm{P}(\bar{M}) = \sum_{M : M_\mathrm{P} = \bar{M}} D(M)$.

4. Security of encryption systems

So far we have only defined the functionality of an encryption system, but not its security. This definition is necessary when arguing about the security of information flow.

Against passive attacks, the most common security definition is the indistinguishability under chosen plaintext attacks (IND-CPA security) (Bellare et al. 1997). It states that there exists no PPT adversary $\mathcal{A}$ (that has access to an oracle), such that the difference of probabilities

$$\Pr[\mathcal{A}^{\mathcal{E}_n(k,\cdot)}(n) = 1 \mid k \leftarrow \mathcal{K}_n()] - \Pr[\mathcal{A}^{\mathcal{E}_n(k,\mathbf{0}^{|\cdot|})}(n) = 1 \mid k \leftarrow \mathcal{K}_n()]$$

is non-negligible. This definition is a typical instance of security definitions of cryptographic primitives and systems where the indistinguishability of the "real" functionality from the "ideal" functionality is required. Here the real functionality is the encryption functionality — given a plaintext it returns a corresponding ciphertext. The ideal functionality also returns a ciphertext, but this ciphertext is generated without actually using the plaintext (except its length).

In the definition of $=_\mathrm{E}$, ciphertexts created with different keys may also be considered equal. Hence we need the encryption system to hide the identities of keys (Abadi and Rogaway 2000) as well: for no PPT adversary $\mathcal{A}$, the following difference may be non-negligible:

$$\Pr[\mathcal{A}^{\mathcal{E}_n(k,\cdot),\mathcal{E}_n(k',\cdot)}(n) = 1 \mid k, k' \leftarrow \mathcal{K}_n()] - \Pr[\mathcal{A}^{\mathcal{E}_n(k,\cdot),\mathcal{E}_n(k,\cdot)}(n) = 1 \mid k \leftarrow \mathcal{K}_n()] \ .$$

Our security proofs in this paper will be simpler if we do not state that in the ideal functionality all keys are the same, but state that in the ideal functionality all keys are different. We also give the adversary the possibility to choose among several encryption oracles, not just two (in the definition of IND-CPA, the adversary could also be allowed to access several encrypting oracles simultaneously).

Let $\mathcal{O}$ be an oracle that works as follows:

- At the initialization (or before it answers its very first query) it independently generates the keys k_i using the algorithm $\mathcal{K}_n$ for all $i \in \mathbb{N}$. Actually, the keys k_i are not all generated in the beginning of the run, but only right before they are first used.
- When queried with (i, x) the machine $\mathcal{O}$ returns $\mathcal{E}_n(k_i, x)$.

Let $\mathcal{O}'$ be an oracle that on query (i, x) generates a new key k, encrypts x with it and returns the result. An encryption system hides the identities of the keys if and only if no PPT adversary is able to distinguish the oracles $\mathcal{O}$ and $\mathcal{O}'$ with non-negligible advantage.

The non-interference definition in the abstract model does not attempt to rule out *key cycles* (Abadi and Rogaway 2000) in any way. A key cycle of length 1 occurs when a key k (or a message where k may be obtained from) is used as a plaintext in encryption with k. In a longer key cycle, the key k_1 is encrypted with k_2, the key k_2 is encrypted with k_3, etc., until the key k_n is encrypted again with k_1. In such scenarios we can find no real functionality, as given in the definition of IND-CPA, that can be replaced with the ideal one. Therefore the definition of IND-CPA does not say anything about the security of such usage of keys. Such usage of keys is certainly possible in programs.

Also, $=_{\mathrm{E}}$ will have no problem relating ciphertexts whose plaintexts are of obviously different lengths. It may make sense to refine $=_{\mathrm{E}}$ so that the lengths of plaintexts were discriminated, but the issue of concealing the lengths is orthogonal to other issues in secure information flow. Hence the current choice should be considered reasonable. A strengthening of the definition of IND-CPA to *key-dependent messages* (Black et al. 2002) that also conceals the lengths of the plaintexts and the identities of the keys can be given as follows.

Let $\mathcal{O}$ be an oracle that works as follows

- At the initialization it independently generates the keys k_i using the algorithm $\mathcal{K}_n$ for all $i \in \mathbb{N}$. Actually, the keys k_i are not all generated in the beginning of the run, but only right before they are first used...
- $\mathcal{O}$ accepts queries of the form (i, e) where $i \in \mathbb{N}$ and e is an expression in some Turing-complete language whose running time is polynomial in n (we may also state that the query contains the maximum running time of e as well). The expression e may contain free variables k_j.
- When queried with (i, e), the machine $\mathcal{O}$ evaluates e, substituting the values of the keys k_j to the free variables k_j of e. It then encrypts the result with the key k_i and returns it.

Let $\mathcal{O}'$ be an oracle that on each query generates a new key and returns the encryption of a fixed constant with this key. The encryption system $(\mathcal{K}, \mathcal{E}, \mathcal{D})$ is *IND-CPA-secure, which-key concealing and length-concealing in presence of key-dependent messages* if no PPT adversary $\mathcal{A}$ can distinguish $\mathcal{O}$ from $\mathcal{O}'$, i.e. the difference of probabilities

$$\Pr[\mathcal{A}^{\mathcal{O}(\cdot)}(n)] - \Pr[\mathcal{A}^{\mathcal{O}'(\cdot)}(n)]$$

must be negligible for all PPT $\mathcal{A}$.

The properties considered above only deal with confidentiality of messages, and only with encryption. They do not state anything about what happens if the decryption algorithm is invoked (the definition of encryption systems states that the decryption of a correctly constructed ciphertext must give back the corresponding plaintext, but does not state anything about decrypting other bit-strings). The property that we are going to need is *plaintext integrity* (INT-PTXT) (Bellare and Namprempre 2000) stating that no PPT adversary $\mathcal{A}$ that is given access to the encryption oracle $\mathcal{E}(k, \cdot)$ (where k is generated using the key generation algorithm) is able (with non-negligible probability) to output a ciphertext c, such that $\mathcal{D}(k, c) = p \neq \perp$ and $\mathcal{A}$ had not queried the encryption oracle with p before. Askarov et al. (2006) argue that if the encryption system has plaintext integrity then $\mathcal{D}(k, \mathcal{E}(k', x))$ results in error almost always (i.e. the opposite has only negligible probability). Here x is any plaintext and k, k' are two keys that are independent of each other.

The above definition of INT-PTXT does not allow encryption cycles, either. We will turn the given definition into one that allows key-dependent messages in the same way as was done for the definition of IND-CPA. First, we give the adversary access to multiple encryption oracles $\mathcal{E}(k_1, \cdot), \mathcal{E}(k_2, \cdot), \ldots$ where the keys k_i are independently generated. Such access is equivalent to the access to an oracle $\mathcal{O}$ that on query (i, x) returns $\mathcal{E}(k_i, x)$. After interacting with the oracle $\mathcal{O}$, the adversary outputs a pair (i, c) and wins if $\mathcal{D}(k_i, c) = p \neq \perp$ and the adversary did not query $\mathcal{O}$ with (i, p) before. The modification of INT-PTXT we have done so far (replacing a single key k with multiple keys $k_1, k_2, \ldots$) has not been substantial — the modified definition can be proved equivalent to the original definition by a standard hybrid argument (Goldreich 2001, Chap. 3.2.3). We will now modify the definition by allowing key-dependent messages as well — in the query (i, x) to $\mathcal{O}$ the component x is not just a bit-string but an expression for computing the bit-string to be encrypted. As before, x may contain free variables $k_1, k_2, \ldots$, these are substituted with the values of the keys $k_1, k_2, \ldots$ that have been generated by $\mathcal{O}$.

Existence. It is not known how to construct encryption systems that are secure in the presence of key-dependent messages, as long as the construction is in the "plain model" i.e. assumes only the existence of one-way functions. Black et al. (2002) give a construction for IND-CPA-secure (with key-dependent messages) encryption system in the *random oracle model* (Bellare and Rogaway 1993). This model assumes the existence of a globally fixed random function $H : \{0,1\}^* \rightarrow \{0,1\}^\omega$ that all parties (including the adversary, and the expressions it sends to its oracle(s)) can access (in practice, H is replaced by some (function based on some) cryptographic hash function). The distribution of H is such that all bits in its image are distributed fairly and independently of each other. In the encryption system by Black et al. (2002), the key generation algorithm just returns an n-bit random string (where n is the security parameter). The encryption algorithm $\mathcal{E}_n(k, x)$ generates an n-bit random string r and returns $(r, H_{|x|}(k \| r) \oplus x)$ where $\|$ denotes the concatenation of bit-strings and H_ℓ returns the first ℓ bits of H. Black et al. (2002) show this system to be IND-CPA-secure. A slight modification of their proof is sufficient to show that this encryption system also conceals the identities of the keys. This encryption system obviously does not hide the length of the plaintext, padding has to be used for that.

Bellare and Namprempre (2000) show how to use message authentication codes (MACs) to provide plaintext (and ciphertext) integrity for encryption systems. A MAC consists of three algorithms

— key generation algorithm, tagging algorithm (taking the key and the message as inputs) and verification algorithm (taking the key, the message and the alleged tag as inputs). The tagging algorithm may be deterministic in which case the verification algorithm simply has to recompute it. A MAC is *weakly unforgeable (WUF)*, if no PPT adversary with the access to tagging and verification algorithms (as oracles) can produce a message and a valid tag, such that the message had not been submitted to the tagging oracle before that. Bellare and Namprempre (2000) prove that a compound encryption system where a message is first tagged with a weakly unforgeable MAC and then encrypted with an IND-CPA-secure cryptosystem (using different keys) provides plaintext integrity. As the encryption operation is the last step in the construction, the construction also preserves the concealing of key identities.

We can modify the definition of weak unforgeability to also allow key-dependent messages (the adversary can then submit expressions, not just messages to the tagging and verification oracles). The proof by Bellare and Namprempre (2000) can then be modified, showing that if the construction is made using primitives that are secure in the presence of key-dependent messages then the resulting encryption system is INT-PTXT-secure also even with key-dependent messages.

A WUF-secure (with key-dependent messages) MAC is easy to construct in the random oracle model. Let the key generation algorithm return a random n-bit string and let the tag of the message x with the key k be the first n bits of $H(k\|x)$. In this way the tags of the messages are really just random bit-strings that do not reveal anything about k.

5. Non-well-structured programs

In this section we give some examples of programs that are secure in the abstract setting, but possibly insecure in the computational setting. We then explain what patterns in programs have to be excluded to make sure that the soundness theorem would not apply to this or analogous programs.

5.1 Bad keys

The program

$$k := s;\ p := \mathsf{enc}(k, s)$$

where $\mathbf{Var}_\mathrm{S} = \{s\}$ and $\mathbf{Var}_\mathrm{P} = \mathbf{Var}_\mathsf{E} = \{p\}$, is secure in the abstract setting. In the computational setting we do not know whether k is a good key. Hence we do not know whether $\mathcal{E}(s, s)$ sufficiently hides s or not. We cannot use the security definitions of encryption systems to argue about the computational security of this program.

The conditions we put on programs must make sure that only keys generated with the operation newkey can be used as keys in encryption operations.

5.2 Constructing ciphertexts

Similarly to using only good keys when encrypting we must use only good ciphertexts when applying the relaxed equivalence $=_\mathrm{E}$ on them.

5.3 Publishing keys

Consider the following program

$$k := \mathsf{newkey};\ p := \mathsf{enc}(k, s)$$

where $\mathbf{Var}_\mathrm{S} = \{s\}$, $\mathbf{Var}_\mathrm{P} = \{k, p\}$ and $\mathbf{Var}_\mathsf{E} = \{p\}$. According to the possibilistic non-interference definition, this program is secure. Indeed, an initial memory $\{s \mapsto v_s\}$ is transformed to the final memories where the value of k is just a key and the part of the value of p that matters for the relation $=_\mathrm{E}$ is just a random initial vector. Hence the public part of the set of final memories does not depend on the initial secret.

On the other hand we would certainly want to consider the preceding program secure if the set of public variables $\mathbf{Var}_\mathrm{P}$ would have consisted of just the variable p. To avoid considering such programs secure we must make sure that the keys cannot become public. This includes using these keys in computations (other than encryption and possibly decryption), such that the results of those computations become public.

Askarov et al. (2006) actually consider the previous program to be insecure because the program state also contains the values of all yet-to-be-generated keys; these values are treated as high-security inputs. Hence one cannot publish keys or anything that depends on them (except when used in encryption).

5.4 Possibilistic vs. probabilistic security

The following well-known example (McLean 1990) shows that the possibilistic non-interference does not always imply probabilistic (and similarly computational) non-interference.

$$x := \mathsf{rnd}(0, 1);\ \mathit{if}\ x\ \mathit{then}\ l := h\ \mathit{else}\ l := \mathsf{rnd}(1, 100)$$

where $\mathsf{rnd}(a, b)$ returns a uniformly chosen random integer between a and b, $l \in \mathbf{Var}_\mathrm{P}$, $h \in \mathbf{Var}_\mathrm{S}$ and it is known that the value of h is between 1 and 100. In the possibilistic setting the semantics of $\mathsf{rnd}(a, b)$ is nondeterministic, we have $\mathsf{rnd}(a, b) \Downarrow^\mathrm{a} c$ for all c where $a \leq c \leq b$.

In the possibilistic setting, the above program is secure because the set of possible final values of l does not depend on the value of h. In the probabilistic setting it is insecure because the most probable final value of l is equal to the value of h. In our programming language, we can implement rnd (or something close to it) by using the probabilistic operation enc. For certain constructions of secure encryption systems, the output of $\mathcal{E}$ is indistinguishable from a uniformly distributed bit-string.

To avoid considering the previous program, we must disallow computations with ciphertexts. The ciphertexts may be published, included in pairs, encrypted or decrypted, but their “actual value” may not be used in computations.

5.5 Non-failure of bad decryptions

Consider the following program:

$$k := \mathsf{newkey};\ k' := \mathsf{newkey};\ x := \mathsf{enc}(k, C);\ y := \mathsf{dec}(k', x);$$
$$\mathit{if}\ y = \perp\ \mathit{then}\ l := h\ \mathit{else}\ l := 1 - h$$

where C is some constant, $\mathbf{Var}_\mathrm{P} = \{l\}$, $\mathbf{Var}_\mathrm{S} = \{h\}$ and the possible values of h are 0 and 1. In the concrete (probabilistic) setting, the program is insecure because the then-branch is taken with overwhelming probability. Still, the encryption system may by such that no matter what the values of k and k' are, there is always a negligible chance that the decryption succeeds[1] and the else-branch is taken. This “non-failure” materializes in the possibilistic setting — both branches are possible. Thus the program is secure — the set of possible values of l does not depend on the value of h.

This particular program is actually insecure by the definitions given by Askarov et al. (2006) because at the failure of the decryption the execution becomes stuck and hence the then-branch is unreachable. But then we could just replace the predicate “$y = \perp$” with some other predicate P that is satisfied by some of the possible values of y and not satisfied by others. Both branches would then still be reachable in the abstract semantics, but in the concrete semantics the branch corresponding to the value of $P(\perp)$ would be chosen most of the time.

[1] Askarov et al. (2006) actually assume that there is no such chance. But it is not clear whether there exist encryption systems that are which-key concealing but where the decryption with a wrong key always fails.

There is also a negligible chance that the two key generations in the considered program return the same key. If this happens then the value of h can also be deduced from the value of l because the *else*-branch is always taken.

We do not have a good remedy against cases like this where the abstract semantics considers the events that happen only negligibly often in the concrete semantics as equals to the events that happen almost always in the concrete semantics. We thus resort to modifying the abstract semantics, such that two key generations cannot produce equal keys, and the decryption of a ciphertext is possible only with the key that was used in creating it. We change the definition of $\Downarrow^{\mathrm{a}}$, it is now a relation of the form $\langle M, K_\circ, C_\circ, e\rangle \Downarrow^{\mathrm{a}} \langle v, K_\bullet, C_\bullet\rangle$ where M is the memory, e the evaluated expression and v its value. Additionally, $K_\circ$ and $K_\bullet$ are the sets of generated keys before and after the evaluation of e, and $C_\circ/C_\bullet$ is the set of triples (c, k, p) of ciphertexts, keys and plaintexts, such that c has been generated by encrypting p with k during the course of the program. The definition of $\Downarrow^{\mathrm{a}}$ for particular expressions is changed as follows:

- For key generations, we have the side condition that the newly generated key v is not a member of $K_\circ$. The set $K_\bullet$ is then defined as $K_\circ \cup \{v\}$ (for all other operations, $K_\bullet = K_\circ$).
- For encryptions, the generated ciphertext may not yet be a ciphertext in the set $C_\circ$. The triple of the newly generated ciphertext, key and plaintext is added to $C_\bullet$ along with the rest of $C_\circ$ (for all other operations, $C_\bullet = C_\circ$).
- The decryption operation $\mathsf{dec}(k, c)$ first computes the plaintext $v_p = \mathcal{D}(M(k), M(c))$. It then checks whether $(v_c, M(k), v_p)$ belongs to $C_\circ$ for some value v_c. If this is the case then $\mathsf{dec}(k, c)$ returns v_p, otherwise it returns $\perp$.

The sets K and C are also added to program configurations.

$$\frac{M(x) = v \quad \Gamma(x) = \tau}{\langle M, \Gamma, K, C, x\rangle \Downarrow^{\mathrm{a}} \langle v, \tau, K, C\rangle}$$

$$\frac{\begin{array}{c}\langle M, \Gamma, K_{i-1}, C_{i-1}, e_i\rangle \Downarrow^{\mathrm{a}} \langle v_i, \mathit{int}, K_i, C_i\rangle \\ [\![o]\!](v_1, \ldots, v_k) = v\end{array}}{\langle M, \Gamma, K_0, C_0, o(e_1, \ldots, e_k)\rangle \Downarrow^{\mathrm{a}} \langle v, \mathit{int}, K_k, C_k\rangle}$$

$$\frac{\langle M, \Gamma, K_{i-1}, C_{i-1}, e_i\rangle \Downarrow^{\mathrm{a}} \langle v_i, \tau_i, K_i, C_i\rangle \quad \rho(v_1, v_2) = v}{\langle M, \Gamma, K_0, C_0, (e_1, e_2)\rangle \Downarrow^{\mathrm{a}} \langle v, (\tau_1, \tau_2), K_2, C_2\rangle}$$

$$\frac{\langle M, \Gamma, K_\circ, C_\circ, e\rangle \Downarrow^{\mathrm{a}} \langle u, (\tau_1, \tau_2), K_\bullet, C_\bullet\rangle \quad \rho(v_1, v_2) = u}{\langle M, \Gamma, K_\circ, C_\circ, \pi_i(e)\rangle \Downarrow^{\mathrm{a}} \langle v_i, \tau_i, K_\bullet, C_\bullet\rangle}$$

$$\frac{\langle M, \Gamma, K_\circ, C_\circ, e\rangle \Downarrow^{\mathrm{a}} \langle u, (\tau_1, \tau_2), K_\bullet, C_\bullet\rangle \quad u \notin \mathrm{Dom}\,\rho}{\langle M, \Gamma, K_\circ, C_\circ, \pi_i(e)\rangle \Downarrow^{\mathrm{a}} \langle \perp, \tau_i, K_\bullet, C_\bullet\rangle}$$

$$\frac{v \in \big(\mathrm{Supp}\,\mathcal{K}()\big)\backslash K}{\langle M, \Gamma, K, C, \mathsf{newkey}\rangle \Downarrow^{\mathrm{a}} \langle v, \mathit{key}, K \cup \{v\}, C\rangle}$$

$$\frac{\begin{array}{c}\langle M, \Gamma, K_0, C_0, e_k\rangle \Downarrow^{\mathrm{a}} \langle v_k, \mathit{key}, K_1, C_1\rangle \\ \langle M, \Gamma, K_1, C_1, e_x\rangle \Downarrow^{\mathrm{a}} \langle v_x, \tau, K_2, C_2\rangle \\ v_y \in \big(\mathrm{Supp}\,\mathcal{E}(v_k, v_x)\big)\backslash\boldsymbol{\pi}_1(C_2) \quad C_3 = C_2 \cup \{(v_y, v_k, v_x)\}\end{array}}{\langle M, \Gamma, K_0, C_0, \mathsf{enc}(e_k, e_x)\rangle \Downarrow^{\mathrm{a}} \langle v_y, \mathit{enc}(\tau), K_2, C_3\rangle}$$

$$\frac{\begin{array}{c}\langle M, \Gamma, K_0, C_0, e_k\rangle \Downarrow^{\mathrm{a}} \langle v_k, \mathit{key}, K_1, C_1\rangle \\ \langle M, \Gamma, K_1, C_1, e_y\rangle \Downarrow^{\mathrm{a}} \langle v_y, \mathit{enc}(\tau), K_2, C_2\rangle \quad v_p = \mathcal{D}(v_k, v_y) \\ v = \mathbf{if}\ (v_k, v_p) \in \boldsymbol{\pi}_{2,3}(C_2)\ \mathbf{then}\ v_p\ \mathbf{else}\ \perp\end{array}}{\langle M, \Gamma, K_0, C_0, \mathsf{dec}(e_k, e_y)\rangle \Downarrow^{\mathrm{a}} \langle v, \tau, K_2, C_2\rangle}$$

where $\boldsymbol{\pi}_{i_1,\ldots,i_k}(C)$ denotes $\{(u_{i_1}, \ldots, u_{i_k}) \mid (u_1, u_2, u_3) \in C\}$

Figure 2. Extended $\Downarrow^{\mathrm{a}}$

6. Well-structured programs

To satisfy the necessary constraints demonstrated by the programs in the previous section, it is sufficient to require that all operations that the program (in the abstract semantics) performs are well-typed. Here the types $\tau \in \mathbf{T}$ are defined by the following grammar:

$$\tau ::= \mathit{int} \mid \mathit{key} \mid \mathit{enc}(\tau) \mid (\tau, \tau),$$

and the operations of the program expect and produce the values of the following types:

- arithmetic operations (of arity k): $\mathit{int}^k \to \mathit{int}$;
- key generation: key;
- encryption: $\mathit{key} \times \tau \to \mathit{enc}(\tau)$;
- decryption: $\mathit{key} \times \mathit{enc}(\tau) \to \tau$;
- pairing: $\tau_1 \times \tau_2 \to (\tau_1, \tau_2)$;
- projections: $(\tau_1, \tau_2) \to \tau_i$.

Also, the guards of the branching statements must have the type int, the secret inputs (i.e. the initial values of the variables in $\mathbf{Var}_{\mathrm{S}}$) must also have the type int, final values of the variables in $\mathbf{Var}_{\mathrm{E}}$ must have types of the form $\mathit{enc}(\tau)$, and final values of the variables in $\mathbf{Var}_{\mathrm{P}}$ may not have a type that has a component key (define that the only component of the types int, key and $\mathit{enc}(\tau)$ is that type itself, and the components of the type (τ_1, τ_2) are the components of τ_1 and τ_2).

The previous paragraph spoke about the types of values, not variables. Indeed, we can allow the typing to be *dynamic*, extending the memories with the information about the current types of variables. This dynamic typing disallows the usage of non-keys as keys and the usage of "actual values" of keys (excluding encryption and decryption) and ciphertexts. To formalize this typing we again extend the definition of $\Downarrow^{\mathrm{a}}$: it is now a set of judgements of the form $\langle M, \Gamma, K_\circ, C_\circ, e\rangle \Downarrow^{\mathrm{a}} \langle v, \tau, K_\bullet, C_\bullet\rangle$, where Γ is a mapping from variables to types and τ is the type of v. The modified $\Downarrow^{\mathrm{a}}$, also covering the modifications of Sec. 5.5, is given in Fig. 2.

The typing Γ is also added to program configurations; the type $\Gamma(x)$ is updated whenever x is assigned to. The extended abstract semantics is given in Fig. 3. We call a program **well-structured** if its execution according to the abstract semantics never gets stuck, i.e. no such configuration is reachable where there is still a program P, but no outgoing transitions. Also, a well-structured program must satisfy the constraints on the final types of variables, as stated above (for variables in $\mathbf{Var}_{\mathrm{P}}$, the type may not be key, and for variables in $\mathbf{Var}_{\mathrm{E}}$, the type must be $\mathit{enc}(\tau)$ for some τ).

The type system still allows us to perform cryptographic operations quite freely, as long as they are not combined with operations that make use of the "actual value" of a key or a ciphertext. We are allowed to copy keys and ciphertexts from one variable to another, as well as to encrypt keys and ciphertexts (or pairs containing a key or a ciphertext as a component). Among the restrictions the inability to use the "actual value" of a ciphertext is the most significant one, but likely unavoidable, because of Sec. 5.4 (otherwise our abstract semantics has to be probabilistic as well). Also, Sec. 5.3 prohibits us making public the values of the keys, hence our type system must ensure that the keys do not end up as values of type int.

LEMMA 1. *Let P be a well-structured program and M a memory. Let $\langle M, \Gamma, K, C, P\rangle \xrightarrow{\mathrm{a}} \langle M', \Gamma', K', C', P'\rangle$. Then P' is well-structured, too.*

$$\frac{\langle M,\Gamma,K_\circ,C_\circ,e\rangle \Downarrow^{\mathrm{a}} \langle v,\tau,K_\bullet,C_\bullet\rangle}{\langle M,\Gamma,K_\circ,C_\circ,x:=e\rangle \xrightarrow{\mathrm{a}} \langle M[x\mapsto v],\Gamma[x\mapsto\tau],K_\bullet,C_\bullet\rangle}$$

$$\frac{}{\langle M,\Gamma,K,C,\mathit{skip}\rangle \xrightarrow{\mathrm{a}} \langle M,\Gamma,K,C\rangle}$$

$$\frac{\langle M,\Gamma,K,C,P_1\rangle \xrightarrow{\mathrm{a}} \langle M',\Gamma',K',C'\rangle}{\langle M,\Gamma,K,C,P_1;P_2\rangle \xrightarrow{\mathrm{a}} \langle M',\Gamma',K',C',P_2\rangle}$$

$$\frac{\langle M,\Gamma,K,C,P_1\rangle \xrightarrow{\mathrm{a}} \langle M',\Gamma',K',C',P_1'\rangle}{\langle M,\Gamma,K,C,P_1;P_2\rangle \xrightarrow{\mathrm{a}} \langle M',\Gamma',K',C',P_1';P_2\rangle}$$

$$\frac{\langle M,\Gamma,K_\circ,C_\circ,e\rangle \Downarrow^{\mathrm{a}} \langle b,\mathit{int},K_\bullet,C_\bullet\rangle \quad b\in\{0,1\}}{\langle M,\Gamma,K_\circ,C_\circ,\mathit{if}\ e\ \mathit{then}\ P_1\ \mathit{else}\ P_0\rangle \xrightarrow{\mathrm{a}} \langle M,\Gamma,K_\bullet,C_\bullet,P_i\rangle}$$

$$\frac{}{\begin{array}{l}\langle M,\Gamma,K,C,\mathit{while}\ e\ \mathit{do}\ P\rangle \xrightarrow{\mathrm{a}} \\ \qquad \langle M,\Gamma,K,C,\mathit{if}\ e\ \mathit{then}\ (P;\mathit{while}\ e\ \mathit{do}\ P)\ \mathit{else}\ \mathit{skip}\rangle\end{array}}$$

Figure 3. Instrumented semantics $\xrightarrow{\mathrm{a}}$

PROOF. Well-structuredness of a program implies the well-structuredness of all of its subprograms. Consider the definition of $\xrightarrow{\mathrm{a}}$. According to it, P' is either a subprogram of P, or $P' \equiv P''; \mathit{while}\ b\ \mathit{do}\ P''$, such that $P \equiv \mathit{while}\ b\ \mathit{do}\ P''$. In the last case P'' and $\mathit{while}\ b\ \mathit{do}\ P''$ are well-structured with respect to the same typing, and their sequential composition admits the same typing as well. □

We can now state the main result of this paper:

THEOREM 2. *If a well-structured program P that does not assign to variables in $\mathbf{Var}_{\mathrm{S}}$ and does not use the initial values of the variables with types different from int has possibilistic non-interference then it has probabilistic non-interference for all families of probability distributions D over initial memories where the variables in $\mathbf{Var}_{\mathrm{S}}$ are independent of the rest of the variables.*

We see that the conditions on P, besides the well-structuredness, are not significant. They can be worked around by adding more variables to P.

7. Security proof

Here we will sketch the proof of Theorem 2. We are going to perform the standard steps in reasoning about programs that contain cryptographic operations — we replace the parts of the program that correspond to the real functionality of the cryptographic primitives with the code that corresponds to the ideal functionality (Sec. 7.1). The modification will transform the program P to some program $\hat{P}$, thereby also changing both the abstract and the concrete semantics of the program. The concrete semantics will change only indistinguishably. We will compare the abstract execution of P with the concrete execution of $\hat{P}$ and conclude that they run in lock-step, producing similar structures of final states (Sec. 7.2). In particular, we will show that for each M, the set of abstract final states $\mathcal{M}$, where $\langle M,P\rangle \stackrel{\mathrm{a}}{\Longrightarrow} \mathcal{M}$, determines the distribution D, where $\langle M,\hat{P}\rangle \stackrel{\mathrm{c}}{\Longrightarrow} D$. If the program P has secure information flow in the abstract setting then the program $\hat{P}$ is probabilistically non-interferent[2]. The concrete semantics of P is indistinguishable from the semantics of $\hat{P}$ for an adversary that does not see the keys generated by the programs, hence P is computationally non-interferent.

[2] The security of $\hat{P}$ is even information-theoretic (Sabelfeld and Sands 1999, Sec. 5), not just computational

before	after
$x := y$	$x := y$
$x := o(x_1,\ldots,x_k)$	$x := (o(\pi_1(x_1),\ldots,\pi_1(x_k)),0)$
$x := \mathsf{newkey}$	$\mathit{idk} := \mathit{idk}+1;$ $x := (\mathsf{newkey},\mathit{idk})$
$x := \mathsf{enc}(k,y)$	$x := (\mathsf{enc}(\pi_1(k),\pi_1(y)),(\pi_2(k),y))$
$y := \mathsf{dec}(k,x)$	$\mathit{if}\ \pi_1(\pi_2(x)) = \pi_2(k)$ $\quad \mathit{then}\ \pi_2(\pi_2(x))\ \mathit{else}\ \perp$
$x := (y,z)$	$x := ((\pi_1(y),\pi_1(z)),(\pi_2(y),\pi_2(z)))$
$y := \pi_i(x)$	$y := (\pi_i(\pi_1(x)),\pi_i(\pi_2(x)))$
$\mathit{if}\ b\ldots$ / $\mathit{while}\ b\ldots$	$\mathit{if}\ \pi_1(b)\ldots$ / $\mathit{while}\ \pi_1(b)\ldots$

Table 1. Removing decryption operations

7.1 Program modifications

We would like to apply the definition of IND-CPA (including the concealing of key identities and plaintext lengths, and security in presence of key-dependent messages), thereby changing the expressions $\mathsf{enc}(k,y)$ to expressions $\mathsf{enc}(\mathsf{newkey},0)$. But we cannot apply this transformation right away — in the definition of IND-CPA the encryption keys may only be used in encryption operations (and they may also be used in arbitrary manner to create the plaintexts that are encrypted), but in our program they may also be used in decryption operations. We have to apply the definition of INT-PTXT first.

The well-structuredness of programs ensures that we only attempt to decrypt valid ciphertexts. We are not ensured that the key used for decryption is the right one. The plaintext integrity of the used encryption system guarantees that if the used key is not the correct one then the decryption almost always fails in the concrete semantics. The modifications made to the abstract semantics in Sec. 5.5 also ensure this for the abstract semantics. We can modify the program so that we record the plaintext of each ciphertext, as well as (the identity of) the key used to create it, and replace the decryption with the return of the plaintext if the comparison of key identities succeeds. This modification does not change the abstract semantics of a program and changes its concrete semantics only indistinguishably.

Let P_0 be the original program. We perform the following program modifications. To simplify the presentation we assume that there are no nested expressions, i.e. in each statement $x := e$ in the program the expression e contains just a single operation. We also assume that all guard expressions in if- and while-statements are just variables. First, we introduce a new variable idk of type int for producing key names and prepend the program with the initialization $\mathit{idk} := 0$. Then we rewrite all assignments and branches according to Table 1.

We see that in the modified program all variables contain pairs whose first component is the original value of the variable. The second component records the auxiliary information for eliminating decryptions:

- for keys, we record their identity;
- for ciphertexts, we record the identity of the key and the plaintext with associated auxiliary information;
- for pairs, we record the auxiliary information of both components;
- we do not record anything for integers, but still keep the second component to avoid special cases in Table 1.

To finish the execution of the program with the same values of variables in $\mathbf{Var}_{\mathrm{P}}$ as before we append to the program the statements $x := \pi_1(x)$ for all $x \in \mathbf{Var}_{\mathrm{P}}$. We also must introduce the second component to all initial values of the variables. For all variables x whose initial values are used by the program we prepend $x := (x, 0)$ to the program (By Thm. 2, all those variables have the initial type int). Let P be the resulting program. Let P' be the program P without the final statements $x := \pi_1(x)$ for $x \in \mathbf{Var}_{\mathrm{P}}$.

LEMMA 3. *If P_0 is a well-structured program then so is P. Moreover, for each initial state the final typings Γ and Γ' of P_0 and P' are such, that $\Gamma'(x) = (\Gamma(x), \ldots)$ for all $x \in \mathbf{Var}$.*

We have introduced the relation $=_{\mathrm{E}}$ on values — two typed values are related by $=_{\mathrm{E}}$ if they are equal or if they are both ciphertexts and have the same initial vector. We now define a more relaxed version of it: we say that $v =_{\mathrm{EK}} v'$ if $v =_{\mathrm{E}} v'$ or both v and v' are keys.

LEMMA 4. *Let M_0 be an initial state. If $\langle M_0, \lambda x.int, \emptyset, \emptyset, P_0\rangle \stackrel{\mathrm{a}}{\Longrightarrow} \mathcal{M}$ and $\langle M_0, \lambda x.int, \emptyset, \emptyset, P'\rangle \stackrel{\mathrm{a}}{\Longrightarrow} \mathcal{M}'$ then for all $M \in \mathcal{M}$ there exists some $M' \in \mathcal{M}'$, and for all $M' \in \mathcal{M}'$ there exists some $M \in \mathcal{M}$, such that $M(x) =_{\mathrm{EK}} M'(\pi_1(x))$.*

The preceding two lemmas can be proved by constructing a (weak) bisimulation between the program configurations of P_0 and P'. The transition relation is $\stackrel{\mathrm{a}}{\longrightarrow}$ in both cases. A configuration of P_0 is related to a configuration of P', if

- the programs correspond to each other;
- the values and types of variables in the configuration of P_0 are related by $=_{\mathrm{EK}}$ to the first components of the values and types of variables in the configuration of P';
- the key identities and recorded plaintexts in the configuration of P' correspond to the recorded ciphertext-key-plaintext triples in the configuration of P_0.

For the concrete semantics we can show the following result.

LEMMA 5. *Let $\mathcal{O}_1$ [resp. $\mathcal{O}_2$] be the following oracle. On input of an initial memory M and the security parameter n, it executes the program P_0 [resp. P] on it (using n as the parameter for the algorithms $\mathcal{K}$, $\mathcal{E}$ and $\mathcal{D}$) and returns the public part of the final memory. Then no PPT algorithm $\mathcal{A}$ can distinguish with non-negligible advantage (in n) whether it interacts with $\mathcal{O}_1$ or $\mathcal{O}_2$.*

Indeed, plaintext integrity of the encryption system ensures that the decryption of a ciphertext with a wrong key returns $\perp$. The proof is again formalized by constructing a weak bisimulation between the program configurations of P_0 and P'. The transition relation is $\stackrel{\mathrm{c}}{\longrightarrow}$ in both cases. The bisimulation relation is similar to the proof of Lemma 4. But this time, the bisimulation is probabilistic, because the transition relation $\stackrel{\mathrm{c}}{\longrightarrow}$ is. Also, the bisimulation is with *error sets* (Backes et al. 2003) — two traces have to be similar only until one of them has reached a state from a certain *error set*. In our case the error sets correspond to situations where a ciphertext can be decrypted by a key different from the one used to create it. Such situation arises with only a negligible probability.

From a well-structured program P_0 we have now constructed a program P that is also well-structured and that is secure in the abstract or concrete setting iff the program P_0 is secure. Also, the program P contains no decryption operations. Hence it suffices to prove Theorem 2 only for programs that contain no decryptions; this, together with the given construction of the program P from the program P_0 immediately implies Theorem 2 in its full generality.

If a program P contains no decryptions then we can apply the definition of IND-CPA-security (the strongest of them in Sec. 4) of the encryption system and replace all expressions $\mathsf{enc}(k, y)$ in P with the expression $\mathsf{enc}(\mathsf{newkey}, 0)$. Let $\hat{P}$ be the resulting program. The concrete semantices of P and $\hat{P}$ are indistinguishable — a result similar to Lemma 5 can be proved for P and $\hat{P}$.

7.2 Similarity of executions

In this subsection we will show that the abstract execution of P and the concrete execution of $\hat{P}$ proceed in some sense in lock-step. Whenever we talk about the concrete semantics of the program $\hat{P}$ in this subsection, the security parameter is implicit. We are going to establish the probabilistic non-interference of $\hat{P}$ without any qualifiers about the power of the adversaries (i.e. the advantage of any adversary is the constant function 0), hence an explicit security parameter would just clutter the notation. We start by noting the following:

LEMMA 6. *Let M be an initial memory and consider an execution $C_0 \stackrel{\mathrm{a}}{\longrightarrow} C_1 \stackrel{\mathrm{a}}{\longrightarrow} \cdots \stackrel{\mathrm{a}}{\longrightarrow} C_{\mathrm{f}}$ where C_0 is the initial configuration $\langle M, \lambda x.int, \emptyset, \emptyset, P\rangle$ and C_{f} is a final configuration. Then the path of that execution through the program P (or: the values of the guard expressions at if- and while-statements) depend only on M, not on the values of keys and ciphertexts generated during the execution.*

Indeed, the guard expressions have to have the type int, but the keys and the ciphertexts have the types key and $enc(\tau)$, respectively, and there are no operations that can be applied to these values that could produce a value of type int (remember that P does not contain decryptions). Similar claims about the keys and ciphertexts not affecting the control flow of the program can be made for the concrete semantics of $\hat{P}$.

Hence for an initial memory M there exists a sequence

$$\langle \mathcal{M}_0, \Gamma_0, P_0\rangle \to \langle \mathcal{M}_1, \Gamma_1, P_1\rangle \to \cdots \to \langle \mathcal{M}_r, \Gamma_r\rangle \qquad (1)$$

where $\mathcal{M}_0 = \{M\}$, $\Gamma_0 = \lambda x.int$, $P_0 = P$, the set of memories $\mathcal{M}_i$ contains all those memories that can be reached by executing P with the initial memory M for i steps, Γ_i is the typing of variables after these i steps and P_i is the program that is still left to execute after i steps. As the path of the execution depends only on M, the typing and the remaining program are the same for all possible memories after i steps. Similarly, for the concrete semantics of $\hat{P}$ there exists a sequence

$$\langle D_0, \hat{P}_0\rangle \to \langle D_1, \hat{P}_1\rangle \to \cdots \to D_{\hat{r}} \qquad (2)$$

where D_0 is the probability distribution that puts all its weight on M, $\hat{P}_0 = \hat{P}$, D_i is the distribution over memories reached after i steps and $\hat{P}_i$ is the program that is still left to execute at this point.

We are going to show that

- $\hat{P}_i$ is the program P_i where all encryption expressions $\mathsf{enc}(k, x)$ are replaced with $\mathsf{enc}(\mathsf{newkey}, 0)$;
- $\langle \mathcal{M}_i, \Gamma_i\rangle$ uniquely determine D_i (without referring to the initial memory M)
  - the mapping from $\langle \mathcal{M}, \Gamma\rangle$ to D will be such that if the public parts of the two sets $\mathcal{M}'$, $\mathcal{M}''$ are equal then the public parts of the corresponding distributions D', D'' are also equal.

these claims imply the probabilistic non-interference of $\hat{P}$. But first we have to explore the structure of the sets $\mathcal{M}_i$ some more.

Given a typing Γ we consider the set of *extended variables* $\mathbf{EVar}_\Gamma$ that contains all "atomic" components of the variables in $\mathbf{Var}$, where "atomic" currently means "having type int, key or $enc(\tau)$". I.e. we want to refer directly to the components of the values of variables whose types are pairs. Formally, $\mathbf{EVar}_\Gamma$ is defined by the following process:

1. Let $\mathbf{EVar}_\Gamma := \mathbf{Var}$.
2. If $\mathbf{EVar}_\Gamma$ contains some z, such that $\Gamma(z) = (\tau_1, \tau_2)$ then
   - Let $\mathbf{EVar}_\Gamma := \mathbf{EVar}_\Gamma \backslash \{z\} \cup \{\pi_1(z), \pi_2(z)\}$;
   - Let $\Gamma := \Gamma[\pi_1(z) \mapsto \tau_1, \pi_2(z) \mapsto \tau_2]$
   - Go to step 2.
3. Otherwise return $\mathbf{EVar}_\Gamma$.

For some $z \in \mathbf{EVar}_\Gamma$ and a memory M whose variables are typed according to Γ we can also define the value of z in M.

LEMMA 7. *Let $\langle \mathcal{M}_i, \Gamma_i, P_i \rangle$ be an element in the sequence (1). Then the following claims hold.*

- *For all extended variables $z \in \mathbf{EVar}_{\Gamma_i}$ with $\Gamma_i(z) = int$, the value of z is the same in all $M \in \mathcal{M}_i$.*
- *There exists a partitioning Π_{K} of the set of all extended variables in $\mathbf{EVar}_{\Gamma_i}$ with type key, such that*
  - *for all $V \in \Pi_{\mathrm{K}}$, $z_1, z_2 \in V$ and $M \in \mathcal{M}_i$, $M(z_1) = M(z_2)$;*
  - *for all $V \in \Pi_{\mathrm{K}}$ and $M_1, M_2 \in \mathcal{M}_i$ there exists $M_3 \in \mathcal{M}_i$, such that*
    - *$M_3(z) = M_1(z)$ for all $z \in V$,*
    - *$M_3(z) =_{\mathrm{E}} M_2(z)$ for all $z \in \mathbf{EVar}_{\Gamma_i} \backslash V$*

    unless some key in M_2 is equal to the keys $M_1(z)$ where $z \in V$.
- *There exists a partitioning Π_{E} of the set of all extended variables in $\mathbf{EVar}_{\Gamma_i}$ with type $enc(\tau)$, such that*
  - *for all $V \in \Pi_{\mathrm{E}}$, $z_1, z_2 \in V$ and $M \in \mathcal{M}_i$, $M(z_1) = M(z_2)$;*
  - *for all $V \in \Pi_{\mathrm{E}}$ and $M_1, M_2 \in \mathcal{M}_i$ there exists $M_3 \in \mathcal{M}_i$, such that*
    - *$M_3(z) =_{\mathrm{E}} M_1(z)$ for all $z \in V$,*
    - *$M_3(z) =_{\mathrm{E}} M_2(z)$ for all $z \in \mathbf{EVar}_{\Gamma_i} \backslash V$*

    unless some ciphertext in M_2 is equal to the ciphertexts $M_1(z)$ where $z \in V$.

The preceding lemma states that at each step of the computation, the values that are present in the variables of the program are the following:

- Integers whose values are fixed.
- A number of keys that may each occur several times. The pattern of copying the keys is fixed. Each key takes all possible values independently of everything else. For example, it cannot happen that the possible values of a key at a certain program point are only half of all possible values for keys, the other half being cut away by some branch statement.
- A number of ciphertexts that may each occur several times. The pattern of copying the ciphertexts is fixed. *The initial vector* of each ciphertext takes all possible values independently of everything else.

Hence the set $\mathcal{M}_i$ is completely determined by Γ_i, the values of extended variables of type int and the partitions Π_{K} and Π_{E}.

The lemma is proved by induction over i, considering all possible steps that a program may make (of which there are just two — assignment and branching). The induction base is $i = 0$, the set $\mathcal{M}_0$ contains just a single memory where all variables have the type int. To simplify the induction step, assume again without lessening of generality that the program contains no nested expressions and that all guard expressions are just variables (the program P constructed in Sec. 7.1 does not satisfy this, so we have to introduce temporary variables to store intermediate results). Assume that the lemma holds for $\mathcal{M}_i$. A branching step is controlled by a guard variable of type int which has the same value in all memories in $\mathcal{M}_i$, hence $\mathcal{M}_{i+1} = \mathcal{M}_i$ in this case. An assignment step $x := e$ can be decomposed into two parts — killing the current value of x and assigning a new value to x. Killing x simply removes the values of all extended variables derived from it from the memory M, thereby possible reducing some sets in the partitions Π_{K} and Π_{E}. The effects of assigning a new value to x depend on e:

- if some values are just copied around (e is a variable, a pair or a projection) then new extended variables of type int will contain a value that is same in all memories, and the equivalence classes of Π_{K} and Π_{E} may be extended with new extended variables;
- if e is $o(x_1, \ldots, x_k)$ then the types of the variables $x_1, \ldots, x_k$ is int, their values are constant across $\mathcal{M}_i$, the operation o is deterministic, and its result is also constant;
- if e is a key generation [resp. encryption] then $\{x\}$ will be a new equivalence class in Π_{K} [resp. Π_{E}]; the possible values of x are all possible keys [resp. have all possible initial vectors].

Recall that P does not contain decryption operations.

We can now define the probability distribution $\mathbf{D}[\mathcal{M}, \Gamma]$ corresponding to a set of memories and a typing satisfying the conditions of Lemma 7 (hence the partitions Π_{K} and Π_{E} are defined). We believe that it is best described informally, by stating how a memory M is constructed when the distribution $\mathbf{D}[\mathcal{M}, \Gamma]$ is sampled. The values of the variables in M have the structure given by Γ — there is the same set of extended variables as in the memories in $\mathcal{M}$. The extended variables of type int have the same values in M as they have in $\mathcal{M}$ — these variables were constants in $\mathcal{M}$ and they are constants in $\mathbf{D}[\mathcal{M}, \Gamma]$. For each equivalence class $V \in \Pi_{\mathrm{K}}$ we generate a key k_V using the algorithm $\mathcal{K}$ and assign the result to all extended variables in V. For each equivalence class $V \in \Pi_{\mathrm{E}}$ we generate a new key k using the algorithm $\mathcal{K}$ and then encrypt the constant 0 with the key k using the algorithm $\mathcal{E}$; the resulting value is assigned to all extended variables in V. Hence the distribution $\mathbf{D}[\mathcal{M}, \Gamma]$ is the "Cartesian product" of a one-point distribution (assigning the values to extended variables of type int), a number of distributions $\mathcal{K}()$ and a number of distributions $\mathcal{E}(\mathcal{K}(), 0)$.

LEMMA 8. *Let $\mathcal{M}_i$, Γ_i and D_i be defined as in (1) and (2). Then $D_i = \mathbf{D}[\mathcal{M}_i, \Gamma_i]$.*

This lemma is again proved by induction over i, considering the possible computation steps. We omit the proof here.

We have shown that if the final sets of memories are the same for the abstract executions from initial memories M_1 and M_2 then the final distributions over memories for the concrete executions from M_1 and M_2 are the same as well. It remains to note that if just the public parts of the final sets of the memories are the same then the public parts of the final distributions are the same as well. The public part of a set of memories $\mathcal{M}$ (together with a typing Γ, such that the conditions of Lemma 7 are satisfied) consists of

- the values of all those extended variables of type int that are parts of the variables in $\mathbf{Var}_{\mathrm{P}}$;
- the initial vectors of the ciphertexts that are the values of all those extended variables of type $enc(\tau)$ that are parts of the variables in $\mathbf{Var}_{\mathrm{P}}$.

The public part of a distribution D over memories is sampled simply by sampling D and only taking the values of variables in $\mathbf{Var}_{\mathrm{P}}$ in the resulting distribution.

The abstract security definition states precisely that modifying only the secret inputs of an initial memory does not change the public part of the resulting final set of memories. Computational non-interference (our security definition in the concrete setting) is a relaxation of probabilistic non-interference stating that the public part of the final distribution may not change if only the secret inputs of the initial memory change. Hence the abstract security of P implies the concrete security of $\hat{P}$ that is equivalent to the concrete security of P.

8. A new model

In this section we give a model that is simpler but equivalent (and sometimes even more permissive) than the semantics and the security definition in the framework for cryptographically masked flows, together with the constraints we gave in Sec. 6. In contrast to the presented abstract model, we can also publish the final values of keys in the new model — there are no constraints of $\mathbf{Var}_{\mathrm{P}}$, except that it must be disjoint with $\mathbf{Var}_{\mathrm{S}}$. Also, the program semantics in the new model is deterministic, in this aspect it is simpler than the model of cryptographically masked flows. The main components of the new model, particularly the equivalence of abstract memories, are similar to the way the equivalence is defined for formal messages by Abadi and Rogaway (2000). It is even more similar to the subsequent development of these results by Abadi and Jürjens (2001) who were also one of the first to use formal randomness to distinguish between different encryptions of the same message with the same key in the abstractions of cryptography (another early paper was (Bodei et al. 2001), but both were influenced by the idea of *confounders* by Abadi (1999)).

We have to redefine the set of values. A value $v \in \mathbf{Val}$ is either $\perp$ or defined by the following grammar:

$$v ::= b \mid \mathsf{k}\langle i\rangle \mid \{v\}^{\mathsf{r}\langle j\rangle}_{\mathsf{k}\langle i\rangle} \mid (v_1, v_2)$$

where $b \in \{0,1\}^*$ and $i, j \in \mathbb{N}$. The value $\mathsf{k}\langle i\rangle$ denotes the (formal) key that is produced by the i-th invocation of newkey. Similarly, $\mathsf{r}\langle j\rangle$ is the formal randomness (or: initial vector) of the j-th ciphertext.

The semantics of operations are functions from tuples of values to values. In particular, the semantics of a normal operation o of arity k is still given by a function from $(\{0,1\}^*)^k$ to $\{0,1\}^*$. If one of the arguments is not a bit-string, the result is $\perp$. The pairing takes two values v_1 and v_2 and returns (v_1, v_2), unless some of v_1, v_2 is $\perp$, in which case it returns $\perp$, too. The projections take the first or second component of a pair; if the argument of a projection is not a pair then the result is $\perp$.

The evaluation context for expressions has to contain the number n_{k} of already generated keys and the number n_{e} of already generated ciphertexts; the values of n_{k} and n_{e} are also part of program configurations. The key generation operation newkey returns $\mathsf{k}\langle n_{\mathrm{k}}+1\rangle$ and increments n_{k}. The encryption operation $\mathsf{enc}(k, y)$ expects the value of k be $\mathsf{k}\langle i\rangle$ for some $i \in \mathbb{N}$. It returns $\{v_y\}^{\mathsf{r}\langle n_{\mathrm{e}}+1\rangle}_{\mathsf{k}\langle i\rangle}$, where v_y is the value of y, and increments n_{e}. If the value of k is not a formal key or if the value of y is $\perp$, it returns $\perp$. The decryption operation expects two arguments of the form $\mathsf{k}\langle i\rangle$ and $\{v\}^{\mathsf{r}\langle j\rangle}_{\mathsf{k}\langle i\rangle}$ and returns v.

Again we require that in the initial memory all values are bit-strings. We define the program P to have secure information flow in our new model if for all memories M_1, M_2, M_1', M_2' where M_i' is the final memory corresponding to the initial memory M_i, $M_1 \sim_{\mathrm{P}} M_2$ implies $M_1' \cong M_2'$. The formal equivalence $\cong$ of memories is defined in the same way as by Abadi and Rogaway (2000). Formally, let $\mathit{visibles}(M) \subset \mathbf{Val}$ be the least set such that

- if $x \in \mathbf{Var}_{\mathrm{P}}$ then $M(x) \in \mathit{visibles}(M)$;
- if $(v_1, v_2) \in \mathit{visibles}(M)$ then $v_i \in \mathit{visibles}(M)$;
- if $\mathsf{k}\langle i\rangle \in \mathit{visibles}(M)$ and $\{v\}^{\mathsf{r}\langle j\rangle}_{\mathsf{k}\langle i\rangle} \in \mathit{visibles}(M)$ then $v \in \mathit{visibles}(M)$.

Let $\mathit{keys}(M) = \mathit{visibles}(M) \cap \{\mathsf{k}\langle i\rangle \mid i \in \mathbb{N}\}$. Extend the set of values by

$$v ::= \ldots \mid \Box^{\mathsf{r}\langle j\rangle},$$

the value $\Box^{\mathsf{r}\langle j\rangle}$ denotes a ciphertext generated using the formal coins $\mathsf{r}\langle j\rangle$, that the adversary is unable to decrypt. Let the *pattern* $\mathit{pat}(v, \mathbf{K})$ of a value v with respect to a set of keys $\mathbf{K}$ be defined as follows:

$$\begin{aligned}
\mathit{pat}(\perp, \mathbf{K}) &= \perp \\
\mathit{pat}(b, \mathbf{K}) &= b \\
\mathit{pat}(\mathsf{k}\langle i\rangle, \mathbf{K}) &= \mathsf{k}\langle i\rangle \\
\mathit{pat}((v_1, v_2), \mathbf{K}) &= (\mathit{pat}(v_1, \mathbf{K}), \mathit{pat}(v_2, \mathbf{K})) \\
\mathit{pat}(\Box^{\mathsf{r}\langle j\rangle}, \mathbf{K}) &= \Box^{\mathsf{r}\langle j\rangle} \\
\mathit{pat}(\{v\}^{\mathsf{r}\langle j\rangle}_{\mathsf{k}\langle i\rangle}, \mathbf{K}) &= \begin{cases} \{\mathit{pat}(v, \mathbf{K})\}^{\mathsf{r}\langle j\rangle}_{\mathsf{k}\langle i\rangle}, & \text{if } \mathsf{k}\langle i\rangle \in \mathbf{K} \\ \Box^{\mathsf{r}\langle j\rangle}, & \text{if } \mathsf{k}\langle i\rangle \notin \mathbf{K} \end{cases}
\end{aligned}$$

Finally define $\mathit{pattern}(M) : \mathbf{Var}_{\mathrm{P}} \to \mathbf{Val}$ by

$$\mathit{pattern}(M)(x) := \mathit{pat}(M(x), \mathit{keys}(M)) \ .$$

We define two memories M_1 and M_2 as *equivalent* if $\mathit{pattern}(M_1)$ and $\mathit{pattern}(M_2)$ are α-conversions of each other. I.e. there must exist permutations $\varphi, \psi : \mathbb{N} \to \mathbb{N}$, such that if we replace each $\mathsf{k}\langle i\rangle$ in $\mathit{pattern}(M_1)$ with $\mathsf{k}\langle \varphi(i)\rangle$ and each $\mathsf{r}\langle j\rangle$ with $\mathsf{r}\langle \psi(j)\rangle$, we get $\mathit{pattern}(M_2)$.

The proof of computational soundness of the secure information flow in the new model is similar to the proof of soundness of cryptographically masked flows. Again we start with the removal of decryption operations by recording the name (identity) of the key alongside each key we generate, and the plaintext and the identity of the key alongside each ciphertext (the change is given in Table 1). There will be no change in the visible abstract semantics.

We can now define the "computational interpretation" of an abstract memory — a mapping from abstract memories to distributions over memories. The computational interpretation is actually identical to the one proposed by Abadi and Rogaway (2000); Adão et al. (2005). We show that the computational interpretation commutes with the computation steps in the abstract and concrete semantics — that the runs in the abstract and in the concrete model proceed in lock-step. Finally we use the result by Adão et al. (2005), stating that formal equivalence of memories implies the indistinguishability of the public parts of their computational interpretations.

Let us see some examples of secure and insecure programs according to the presented definition. Consider the following program

$$\begin{aligned}
k := \mathsf{newkey};\ \mathit{if}\ h\ \ \mathit{then}\ l_1 := \mathsf{enc}(k, a);\ l_2 := \mathsf{enc}(k, b) \\
\mathit{else}\ l_2 := \mathsf{enc}(k, a);\ l_1 := \mathsf{enc}(k, b)
\end{aligned}$$

where $\mathbf{Var}_{\mathrm{S}} = \{h\}$ and $\mathbf{Var}_{\mathrm{P}} = \{l_1, l_2\}$. The quantities a and b are constants. Depending on the initial value of h (either 0 or 1), the final memories of this program will be

$$\{h \mapsto 1, k \mapsto \mathsf{k}\langle 1\rangle, l_1 \mapsto \{a\}^{\mathsf{r}\langle 1\rangle}_{\mathsf{k}\langle 1\rangle}, l_2 \mapsto \{b\}^{\mathsf{r}\langle 2\rangle}_{\mathsf{k}\langle 1\rangle}\}$$

and

$$\{h \mapsto 0, k \mapsto \mathsf{k}\langle 1\rangle, l_1 \mapsto \{b\}^{\mathsf{r}\langle 2\rangle}_{\mathsf{k}\langle 1\rangle}, l_2 \mapsto \{a\}^{\mathsf{r}\langle 1\rangle}_{\mathsf{k}\langle 1\rangle}\} \ .$$

The patterns of the public parts are

$$\{l_1 \mapsto \Box^{\mathsf{r}\langle 1\rangle}, l_2 \mapsto \Box^{\mathsf{r}\langle 2\rangle}\}$$

and

$$\{l_1 \mapsto \Box^{\mathsf{r}\langle 2\rangle}, l_2 \mapsto \Box^{\mathsf{r}\langle 1\rangle}\} \ .$$

We see that the patterns do not depend on the constants a and b. Furthermore, the first of them can be α-converted to the second one by letting the permutation ψ of formal randomness indices map 1 to 2 and 2 to 1. Hence the considered program is secure.

Consider now the following program:

$$k := \mathsf{newkey};\ l_1 := \mathsf{enc}(k, a);\ \mathit{if}\ h\ \ \mathit{then}\ l_2 := \mathsf{enc}(k, b)\ \mathit{else}\ l_2 := l_1$$

with the same $\mathbf{Var}_\mathrm{S}$, $\mathbf{Var}_\mathrm{P}$, a and b as before. Depending on the value of h, the final memories of this program will be

$$\{h \mapsto 1, k \mapsto \mathsf{k}\langle 1\rangle, l_1 \mapsto \{a\}^{\mathsf{r}\langle 1\rangle}_{\mathsf{k}\langle 1\rangle}, l_2 \mapsto \{b\}^{\mathsf{r}\langle 2\rangle}_{\mathsf{k}\langle 1\rangle}\}$$

and

$$\{h \mapsto 0, k \mapsto \mathsf{k}\langle 1\rangle, l_1 \mapsto \{a\}^{\mathsf{r}\langle 1\rangle}_{\mathsf{k}\langle 1\rangle}, l_2 \mapsto \{a\}^{\mathsf{r}\langle 1\rangle}_{\mathsf{k}\langle 1\rangle}\}\ .$$

The patterns of the public parts are

$$\{l_1 \mapsto \Box^{\mathsf{r}\langle 1\rangle}, l_2 \mapsto \Box^{\mathsf{r}\langle 2\rangle}\}$$

and

$$\{l_1 \mapsto \Box^{\mathsf{r}\langle 1\rangle}, l_2 \mapsto \Box^{\mathsf{r}\langle 1\rangle}\}\ .$$

As there exists no α-conversion between these possible patterns, the program is not secure. Indeed, the value of h can be determined by considering the equality of l_1 and l_2.

9. Related work

Cryptographically masked flows were proposed by Askarov et al. (2006) along with a type system for checking the security of information flow. Askarov and Sabelfeld (2007) considered the release of keys in this framework, using it to provide a mechanism for declassification.

In quite general terms, the topic of this paper is secure information flow. A semi-recent overview of this topic is given by Sabelfeld and Myers (2003). The handling of encryption and publishing the ciphertexts is also closely connected to the topic of declassification, a good overview of which is given by Sabelfeld and Sands (2005).

In this paper we have searched for *abstractions* of cryptography; we have been particularly concerned with the soundness of such abstractions. The most well-known and celebrated abstraction of cryptography is without doubt the Dolev-Yao model (Dolev and Yao 1983) that abstracts the cryptographic messages by terms in a free algebra and lists all possible means to derive new messages from the known ones. The Dolev-Yao model even appears in this paper, in Sec. 8. The question on the soundness has been unanswered for a long time, the first well-known results connecting it to the computational model were given by Abadi and Rogaway (2000). Abadi and Rogaway (2000) considered only passive adversaries, but the soundness (for integrity properties) in the presence of active adversaries was shown by Cortier and Warinschi (2005). Another, earlier soundness proof was given by Herzog (2002), but this proof required plaintext aware encryption systems for which no constructions without random oracles are known. Also, a comprehensive soundness result was established by the presentation of the universally composable cryptographic library (Backes et al. 2003) showing that minor adjustments to the Dolev-Yao model indeed cause it to reflect all observable properties of the computational model. Still, those results about the Dolev-Yao model do not carry that easily over to cryptographically masked flows.

Recently, some other abstractions of the encryption functionality have appeared, although the focus of these works has been different — an information-hiding and integrity-preserving construction has been proposed and then it is shown how to implement it using cryptographic primitives. Vaughan and Zdancewic (2007) have proposed an *information-packing* primitive that declassifies its argument from its original security level to the lowest level, but at the same time preserves its confidentiality and integrity as if it had not been declassified. The packing primitive is integrated into the decentralized label model (Myers 1999) and implemented using public-key encryption and signatures. Fournet and Rezk (2008) give a sound implementation of the shared memory security model (stating which principal is allowed to read or write which variables) using cryptography (again, public-key encryption and signatures). The soundness of the implementation is shown using language-based techniques, by giving type systems for secure information flow for both the source and target languages (which may be of independent interest) and showing that the translation preserves typing.

10. Discussion

This paper has demonstrated the limits of cryptographically masked flows. Some of them are quite natural (e.g. the handling of keys), but one of them in particular may seem excessive, partly because the program analyses working directly on the computational semantics (Laud 2001, 2003; Laud and Vene 2005) do not have that limit. We are concerned about the inability to use the "actual values" of ciphertexts in arbitrary computations.

Still, as Sec. 5.4 demonstrated, we likely cannot allow the arbitrary handling of random values (in the concrete semantics), unless these values are somehow randomly generated in the abstract semantics as well. Models where the encryption is replaced with random number generation have been considered (Smith and Alpízar 2006) and they would be quite similar to the programs that we get after performing the replacement of the real functionality of the encryption primitive with the ideal functionality. If we also want to have decryption operations in such a model then we have to keep a table of plaintext-key-ciphertext pairs that have occurred in the course of the computation, similarly to the modifications of the abstract semantics in Sec. 5.5. The practical value of such abstract model is strongly dependent on the available tool support. Fortunately, tools for arguing about stochastic programs will suffice; the tools do not have to deal with the cryptographic effects because these have been abstracted away.

11. Acknowledgements

This research has been supported by Estonian Science Foundation, grant No. 6944 and by EU Integrated Project MOBIUS (contract no. IST-15905). We thank the anonymous referees and Steve Zdancewic for their valuable comments, as well as Pierpaolo Degano for pointing out the origins of formal randomness.

References

Martín Abadi. Secrecy by Typing in Security Protocols. *Journal of the ACM*, 46(5):749–786, September 1999.

Martín Abadi and Jan Jürjens. Formal Eavesdropping and Its Computational Interpretation. In Naoki Kobayashi and Benjamin C. Pierce, editors, *Theoretical Aspects of Computer Software, 4th International Symposium, TACS 2001*, volume 2215 of *LNCS*, pages 82–94, Sendai, Japan, October 2001. Springer-Verlag.

Martín Abadi and Phillip Rogaway. Reconciling Two Views of Cryptography (The Computational Soundness of Formal Encryption). In Jan van Leeuwen, Osamu Watanabe, Masami Hagiya, Peter D. Mosses, and Takayasu Ito, editors, *International Conference IFIP TCS 2000*, volume 1872 of *LNCS*, pages 3–22, Sendai, Japan, August 2000. Springer-Verlag.

Pedro Adão, Gergei Bana, Jonathan Herzog, and Andre Scedrov. Soundness of formal encryption in the presence of key-cycles. In Sabrina De Capitani di Vimercati, Paul F. Syverson, and Dieter Gollmann, editors, *ESORICS*, volume 3679 of *Lecture Notes in Computer Science*, pages 374–396. Springer, 2005.

Johan Agat. Transforming out timing leaks. In *POPL 2000, Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 40–53, Boston, Massachusetts, January 2000. ACM Press.

Aslan Askarov and Andrei Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In Pfitzmann and McDaniel (2007), pages 207–221.

Aslan Askarov, Daniel Hedin, and Andrei Sabelfeld. Cryptographically-Masked flows. In Kwangkeun Yi, editor, *SAS*, volume 4134 of *Lecture Notes in Computer Science*, pages 353–369. Springer, 2006.

Michael Backes, Birgit Pfitzmann, and Michael Waidner. A Universally Composable Cryptographic Library. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, Washington, DC, October 2003. ACM Press. Extended version available as Report 2003/015 of Cryptology ePrint Archive.

Mihir Bellare and Chanathip Namprempre. Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm. In Tatsuaki Okamoto, editor, *Advances in Cryptology - ASIACRYPT 2000, 6th International Conference on the Theory and Application of Cryptology and Information Security*, volume 1976 of *LNCS*, pages 531–545, Kyoto, Japan, December 2000. Springer-Verlag.

Mihir Bellare and Phillip Rogaway. Random Oracles are Practical: A Paradigm for Designing Efficient Protocols. In *CCS '93, Proceedings of the 1st ACM Conference on Computer and Communications Security*, pages 62–73, Fairfax, Virginia, November 1993. ACM Press.

Mihir Bellare, Anand Desai, Eron Jokipii, and Phillip Rogaway. A Concrete Security Treatment of Symmetric Encryption. In *38th Annual Symposium on Foundations of Computer Science*, pages 394–403, Miami Beach, Florida, October 1997. IEEE Computer Society Press.

John Black, Phillip Rogaway, and Thomas Shrimpton. Encryption-scheme security in the presence of key-dependent messages. In Kaisa Nyberg and Howard M. Heys, editors, *Selected Areas in Cryptography*, volume 2595 of *Lecture Notes in Computer Science*, pages 62–75. Springer, 2002.

Chiara Bodei, Pierpaolo Degano, Flemming Nielson, and Hanne Riis Nielson. Static analysis for secrecy and non-interference in networks of processes. In Victor E. Malyshkin, editor, *PaCT*, volume 2127 of *Lecture Notes in Computer Science*, pages 27–41. Springer, 2001.

Veronique Cortier and Bogdan Warinschi. Computationally Sound, Automated Proofs for Security Protocols. In Shmuel Sagiv, editor, *Programming Languages and Systems, 14th European Symposium on Programming, ESOP 2005*, volume 3444 of *LNCS*, pages 157–171, Edinburgh, UK, April 2005. Springer-Verlag.

Danny Dolev and Andrew C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT-29(12):198–208, March 1983.

Cédric Fournet and Tamara Rezk. Cryptographically Sound Implementations for Typed Information-Flow Security. In *POPL 2008, Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Francisco, California, January 2008. ACM Press.

Joseph A. Goguen and José Meseguer. Security Policies and Security Models. In *Proceedings of the 1982 IEEE Symposium on Security and Privacy*, pages 11–20, Oakland, California, April 1982. IEEE Computer Society Press.

Oded Goldreich. *Foundations of Cryptography. Volume 1 - Basic Tools*. Cambridge University Press, 2001.

Jonathan Herzog. Computational Soundness of Formal Adversaries. Master's thesis, Massachusetts Institute of Technology, September 2002.

Peeter Laud. Semantics and Program Analysis of Computationally Secure Information Flow. In David Sands, editor, *Programming Languages and Systems, 10th European Symposium on Programming, ESOP 2001*, volume 2028 of *LNCS*, pages 77–91, Genova, Italy, April 2001. Springer-Verlag.

Peeter Laud. Handling Encryption in Analyses for Secure Information Flow. In Pierpaolo Degano, editor, *Programming Languages and Systems, 12th European Symposium on Programming, ESOP 2003*, volume 2618 of *LNCS*, pages 159–173, Warsaw, Poland, April 2003. Springer-Verlag.

Peeter Laud and Varmo Vene. A Type System for Computationally Secure Information Flow. In Maciej Liśkiewicz and Rüdiger Reischuk, editors, *15th International Symposium on Fundamentals of Computation Theory (FCT) 2005*, volume 3623 of *LNCS*, pages 365–377, Lübeck, Germany, August 2005. Springer-Verlag.

John McLean. Security models and information flow. In *IEEE Symposium on Security and Privacy*, pages 180–189, 1990.

Andrew C. Myers. JFlow: Practical Mostly-Static Information Flow Control. In *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 228–241, San Antonio, Texas, January 1999. ACM Press.

Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: A Formal Introduction*. Wiley, 1992.

Birgit Pfitzmann and Patrick McDaniel, editors. *2007 IEEE Symposium on Security and Privacy (S&P 2007), 20-23 May 2007, Oakland, California, USA*, 2007. IEEE Computer Society.

Andrei Sabelfeld and Andrew C. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

Andrei Sabelfeld and David Sands. A Per Model of Secure Information Flow in Sequential Programs. In S. Doaitse Swierstra, editor, *Programming Languages and Systems, 8th European Symposium on Programming, ESOP'99*, volume 1576 of *LNCS*, pages 40–58, Amsterdam, The Netherlands, March 1999. Springer-Verlag.

Andrei Sabelfeld and David Sands. Dimensions and principles of declassification. In *CSFW*, pages 255–269. IEEE Computer Society, 2005.

Geoffrey Smith and Rafael Alpízar. Secure Information Flow with Random Assignment and Encryption. In *4th ACM Workshop on Formal Methods in Security Engineering*, pages 33–43, 2006.

Geoffrey Smith and Dennis M. Volpano. Secure Information Flow in a Multi-threaded Imperative Language. In *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 355–364, San Diego, California, January 1998. ACM Press.

Jeffrey A. Vaughan and Steve Zdancewic. A cryptographic decentralized label model. In Pfitzmann and McDaniel (2007), pages 192–206.