Information Flow Security of Multi-Threaded Distributed Programs

Riccardo Focardi Matteo Centenaro

Dipartimento di Informatica
Università Ca' Foscari di Venezia
{focardi,centenaro}@dsi.unive.it

Abstract

We study noninterference in the setting of multi-threaded distributed programs in which threads share local memories and multi-threaded processes communicate over an insecure network using encryption primitives to secure messages. We extend a simple imperative language with cryptographic operations which are modelled as special expressions respecting the Dolev-Yao assumptions. Then, we adapt to our setting the notion of patterns proposed by Abadi and Rogaway for modelling the equivalence of cryptographic expressions. Based on this notion, we naturally obtain a definition of strongly secure programs corresponding to the one proposed by Sabelfeld and Sands for programs without cryptography. This is, to the best of our knowledge, the first definition of noninterference in a multi-threaded distributed setting, with insecure channels and cryptography. We prove compositionality of secure programs and we adapt the type system of Sabelfeld and Sands to our setting, proving its correctness.

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

General Terms Security, Languages, Verification, Theory

1. Introduction

Protecting confidentiality of data stored in a computer system or transmitted on a network, is a relevant issue in computer security. Secure information flow analysis aims at preventing programs from leaking sensitive data. This can be done via static program analysis, as first observed by Denning and Denning [8], i.e., by inspecting applications' source code in order to point out any potential insecurity. The *noninterference* property (NI) [9] plays a crucial role in secure information flow analysis and has been deeply studied in the last years, as the survey by Sabelfeld and Myers shows [14]. In the simplest case two information confidentiality levels are taken into account: secret (*high*) and public (*low*). Intuitively, NI ensures that computation results provide a low level view of the system which is independent of the high data, i.e., that *high* does not interfere with *low*.

Our goal, here, is to study noninterference in the setting of multi-threaded distributed programs in which threads share local memories and multi-threaded process communicate over an insecure network. Due to the public nature of the network, programs will use encryption primitives to secure messages.

We employ a simple imperative while-language to develop our analysis. Cryptographic operations are modelled as special expressions respecting the Dolev-Yao assumptions, i.e., an attacker can never decrypt a ciphertext without knowing the right decryption key. As observed in [5, 17], studying information flow in this setting is challenging because common NI definitions forbid any flow from private to public data, while encrypting high data with a high level key is expected to produce a low result that might be sent on the insecure network, like in

$$\mathsf{send}(\mathsf{encrypt}(h, k))$$

where h and k are, respectively, a high level datum and a high key. Since the encryption depends on high level, variations of h cause variations on the ciphertext which become observable when the message is sent over the network. This program would be thus judged as insecure by standard NI notions.

We propose a new variant of NI which correctly deals with cryptographic messages in the Dolev-Yao model. First of all, since information on plaintexts can be obtained by comparing the corresponding ciphertexts, we adopt a form of randomized encryption based on unpredictable confounders, similarly to what is done in [1]. The idea is that each encryption contains a fresh confounder which makes it different from every previous and future encryption. Then, we give a new notion of memory/network equivalence, based on the notion of patterns proposed by Abadi and Rogaway [3] and

PLAS'08, June 8, 2008, Tucson, Arizona, USA.

extended by Abadi and Jürjens [2], which is able to capture the possibility for an intruder of observing different instances of the *same* encryption. In fact, encrypting twice the same message will produce different ciphertexts, but copying the same encryption twice will produce the very same encrypted messages like in the following example program taken from [11]:

$$\begin{array}{l} l_1 := \mathsf{encrypt}(h_1, k) \\ \mathsf{if}\ h\ \mathsf{then}\ l_2 := \mathsf{encrypt}(h_1, k) \\ \quad \mathsf{else}\ l_2 := l_1 \end{array} \qquad (1)$$

Depending on the high level variable h, we assign to l_2 either a new encryption of h_1 with k or the encrypted value stored in l_1. Because of confounders, the assigned values will differ and, only in the else branch, we will have that $l_1 = l_2$, thus making the boolean h observable.

This new equivalence notion, apart from incorporating the above mentioned Dolev-Yao assumption, looks for possible *patterns* of equal encrypted messages and requires that those patterns are the same so to make it impossible for the intruder to achieve any information about secret encrypted data.

Based on this notion, we naturally obtain a definition of *strongly secure programs* corresponding to the one proposed by Sabelfeld and Sands [15] for programs without cryptography. This is, to the best of our knowledge, the first definition of noninterference in a multi-threaded distributed setting, with insecure channels and cryptography. Interestingly, we only refine the underlying notion of low-equivalent states leaving the remaining part of the definition, i.e., the low-bisimulation, substantially the same. This minimal change, together with the fact that cryptography is modelled via expressions, simplifies the task of re-proving known results. In particular, we prove compositionality of secure programs and we adapt the type system of Sabelfeld and Sands [15] to our setting, proving its correctness.

Related work. Information flow for programs that employ explicit cryptographic operations has been studied recently by Askarov, Hedin and Sabelfeld [5], Smith and Alpìzar [17] and Laud [11]. All of these papers, however, propose models and properties for sequential programs without multi-threading or concurrency.

More specifically, in [5] the authors adopt the notion of possibilistic noninterference, a weaker variant of noninterference. This choice has been driven by the need of distinguishing between different encryptions and copies of the same ciphertexts, as we have illustrated with Program (1) above. The limitation of such a notion, however, is that it does not deal with possible concurrent thread executions. Consider, for example, the following program:

$$\begin{array}{l} h := \mathsf{true} \\ \mathsf{if}\ (h)\ \mathsf{then}\ l := \mathsf{true} \\ \quad \mathsf{else}\ l := \mathsf{false} \end{array}$$

It is clear that in a single-threaded setting this code can be referred as secure: in fact using the (possibilistic) noninterference notion of [5] the program would be considered secure (even if it would be rejected by the type system). Intuitively, such a property observes the result after program termination which, independently of the initial values of h and l, is always h : true, l : true. Our definition rejects such a program because a thread running together with the above code could change the value stored on the secret h just before the if command, thus making the program change its execution path and reveal the new high value.

The work by Smith and Alpìzar [17] uses computational probabilistic noninterference on a language with random assignments. The language is not multi-threaded but random assignments break the determinism of sequential programs making the setting much more complicate than just single-threading. The paper focus on the computational counterpart of noninterference, that we instead do not consider here.

Another very recent paper on this line of research is [11], where Laud investigates conditions under which the model proposed in [5] is computationally sound. The author essentially proves a conjecture made in [5] about the properties required on the underlying cryptographic primitives to guarantee computational security for programs that satisfy the possibilistic noninterference property discussed above. Interestingly, at the end of the paper, Laud suggests a variant of the model of [5] based on the same definition of patterns we adopt in this paper [3, 2]. He still employs possibilistic noninterference for a single-threaded language. For example, consider the following example taken from [11]:

$$\begin{array}{l} k := \mathsf{newkey} \\ \mathsf{if}\ (h)\ \mathsf{then} \\ \quad l_1 := \mathsf{encrypt}(a, k) \\ \quad l_2 := \mathsf{encrypt}(b, k) \\ \mathsf{else} \\ \quad l_2 := \mathsf{encrypt}(a, k) \\ \quad l_1 := \mathsf{encrypt}(b, k) \end{array}$$

Notice that the order of assignments is swapped in the two branches. Nevertheless, Laud's model accept this program as secure, given that, at the end of execution, the two low variables are assigned to two different (randomized) ciphertexts. In a multi-threaded environment, however, we can think of the intruder as a concurrent thread observing, step by step, the program execution (and possibly controlling the scheduling). It is clear that by observing which of the two variables is assigned first, the intruder can deduce the value of h. Our notion of noninterference correctly rejects this program. In a previous work, Laud [10] presented a type system to check secrecy of messages in cryptographic protocols implementation. While addressing multi-threading it was not aiming at noninterference result.

Vaughan and Zdancewic [19] study interaction between cryptography and information flow using implicit primitives in a single-threaded imperative language and obtaining a

noninterference result which is based on both static and dynamic checking. They implement a decentralized label model (DLM) where confidentiality and integrity requirements can be specified independently. Chothia, Duggan and Vitek [7] first investigated the combination of DLM-style policies and cryptography but without providing any noninterference result.

Finally, papers [12, 13] are also quite related with ours, even if they do not treat explicit cryptography. In particular, the language for distributed multi-threaded programs we adopt here derives from the one proposed in those papers. Differently from [12, 13] we only consider insecure channels and, consequently secret data needs to be encrypted before being sent over the network. The noninterference property we adopt is basically the same but, because of the cryptographic messages, the underlying low-equivalence notion is completely different, as already discussed.

The paper is organized as follows: Section 2 presents the language, Section 3 gives the NI notion of *strongly secure programs* showing it is inadequate for dealing with cryptography; Section 4 illustrates the new NI notion, through many simple examples; Section 5 presents some results about composition of secure programs; Section 6 gives a type system for the proposed NI property; Section 7 draws some concluding remarks.

For space reason we are forced to omit most of the proofs which can be found in the full version [6].

2. A Language with Cryptography

Our language is an extension with explicit cryptography of the Multi-Threaded While-Language with Message Passing of [12], a simple imperative language with message passing communication. Instead of assuming the presence of secure channels, as done in [12], we assume channels are all public and thus accessible by every program. Security is then achieved by explicit cryptographic operations which we model via the special expressions encrypt and decrypt. For the sake of readability, we only consider a synchronous, blocking, receive. We are confident all the results will scale to the full language of [12], in which a non-blocking if-receive is considered, too. As in the original language, our send is asynchronous. The language syntax follows:

$$
\begin{array}{rcl}
C,D & ::= & \mathsf{skip} \mid var := Exp \mid C_1; C_2 \\
& \mid & \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \mid \mathsf{while}\ B\ \mathsf{do}\ C \\
& \mid & \mathsf{fork}(C\vec{C}) \mid \mathsf{send}(cid, Exp) \mid \mathsf{receive}(cid, var) \\
Exp & ::= & var \mid \mathsf{op}(Exp_1, \ldots, Exp_n)
\end{array}
$$

Commands CMD are ranged over by C,D, while $\vec{C}, \vec{D}$ denote possibly empty vectors of concurrent commands $\langle C_1\ C_2\ \ldots\ C_n\rangle$, representing *multi-threaded programs*, variables var range over VAR, expressions Exp range over EXP, boolean expressions B range over $BOOL$, values v range over VAL and channel identifiers cid range over a fixed set CID. Variables VAR are partitioned into two disjoint sets $\mathcal{H}$ and $\mathcal{L}$ of high and low variables, ranged over by h and l. The commands $\mathsf{send}(cid,Exp)$ and $\mathsf{receive}(cid,var)$ are used to send and receive messages on network channel identified by cid. As already mentioned, channels are all public (in contrast to channels partition assumed in [12]), i.e., every program can access them, and are accessed with a standard First-In-First-Out (FIFO) policy. We model cryptographic operations as special expressions following the Dolev-Yao assumptions, as explained below.

Language Semantics A *configuration* $\langle\!|\, \vec{C}, m, \sigma \,|\!\rangle$ is a triple consisting of a vector of commands $\vec{C}$, a local memory m and network state σ. The memory $m : VAR \to VAL$ is a finite map from variables to values. The network state $\sigma : CID \to LVAL$ returns, for each channel identifier, the ordered list of values which are present on that channel. A program shares, over its threads, the local memory m and we suppose that executions happen on a single processor, i.e., at most one thread is active at a given point in time. Distributed programs $\vec{C_1}, \ldots \vec{C_n}$ have their own memories $m_1, \ldots, m_n$ and communicate via the network whose state is represented by σ. *Global configurations*, noted $\lhd(\vec{C_1}, m_1), \ldots, (\vec{C_n}, m_n); \sigma \rhd$, represent distributed programs.

The semantics is formalized in Table 1 by transitions between configurations and global configurations. In particular, $\twoheadrightarrow$ transition formalizes the deterministic execution of sequential commands. Intuitively, skip does nothing; $var := Exp$ assigns the evaluation of Exp to variable var (notice that expressions are atomically evaluated using the function $\downarrow^m$ which returns $\perp$ in case of failure, as described below); $C_1; C_2$ is the sequential composition of C_1 and C_2, in which possible generation of threads $\vec{D}$ performed by C_1 is executed concurrently, as expected; $\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$ executes C_1 or C_2 depending on whether B evaluates to true or not and $\mathsf{while}\ B\ \mathsf{do}\ C$ is handled similarly; $\mathsf{fork}(C\vec{C})$ dynamically generates a new vector of threads $\vec{C}$ running in parallel with C; $\mathsf{send}(cid,Exp)$ and $\mathsf{receive}(cid,var)$ respectively sends and receives values on the channel cid which is modeled as a FIFO queue. In [12], channels are modelled as unordered lists thus capturing a lower level view of distributed systems in which the order of message delivery is not guaranteed. We can easily adapt our definitions to also deal with this different assumption.

Concurrency is modeled by transitions $\to$ and $\twoheadrightarrow$, the first non-deterministically picking a thread and executing it via $\twoheadrightarrow$, the second non-deterministically picking a multi-threaded program and executing it via $\to$. Intuitively, $\to$ acts as purely nondeterministic scheduler among all the threads, while $\twoheadrightarrow$ gives an interleaving semantics to the global distributed system.

Cryptography We model cryptography via special expressions. In particular, in the set of expressions EXP, we as-

sume to have all the usual arithmetic and relational expressions plus encrypt and decrypt for cryptography, pair for the constructions of tuples, fst and snd to access their contents. We thus consider the following values, ranged over by v:

$$v ::= \bot \mid n \mid b \mid \{v\}_n \mid (v_1, v_2)$$

where $\bot$ is a special value representing failures, n denotes a generic atomic value, b ranges over booleans, $\{v\}_n$ represents the encryption of v using n as key and (v_1, v_2) is a pair of values. We will sometimes omit the brackets to simplify the notation, e.g., we will write $\{v_1, v_2\}_n$ for $\{(v_1, v_2)\}_n$.

Based on this set of values we can give the semantics of the special expressions mentioned above:

$$\begin{array}{rcll} \mathsf{encrypt}(v, n) & = & \{v, c\}_n & c \leftarrow \mathcal{C} \\ \mathsf{decrypt}(\{v, c\}_n, n) & = & v & \\ \mathsf{newkey} & = & k & k \leftarrow KEY \\ \mathsf{pair}(v_1, v_2) & = & (v_1, v_2) & \\ \mathsf{fst}((v_1, v_2)) & = & v_1 & \\ \mathsf{snd}((v_1, v_2)) & = & v_2 & \end{array}$$

where c is a fresh confounder, i.e., a number which is used for one encryption and never used again, and k is a fresh key. The notation $\leftarrow$ is used to represent random extraction from a set of values, namely $\mathcal{C}$ is the stream of confounders and KEY the one of encryption keys. Further details on this latter stream will be given later on. The probability of extracting the same random value is negligible, if the set is suitably large, so we actually model random extraction by requiring that extracted values are always different, e.g., $c \leftarrow \mathcal{C}$ can be thought as extracting the first element of an infinite stream of confounders and removing it from the list so that it cannot be reused. More formally we assume that two extractions $c, c' \leftarrow \mathcal{C}$ are such that $c \neq c'$. A similar solution is also adopted, e.g., in [1, 2].

Interestingly, the above functions are not defined for all the possible values. For example, decrypting with the wrong key is undefined, as is taking the first element of a value which is not a pair. We assume that all the undefined cases will fail producing a $\bot$ as result. This choice will influence command semantics, as described below. A simpler solution would be to stop execution for the undefined cases. This would however make many programs insecure if we assume the intruder is able to observe termination (as we will). For example the following program reads a message from the network, decrypts it using a secret key k, then sends out a public value.

$$\begin{array}{l} \mathsf{receive}(cid, x) \\ y := \mathsf{decrypt}(x, k) \\ \mathsf{send}(cid, l) \end{array}$$

If decrypt stopped the execution then the last message would not be sent and the intruder could gain information about the message sent to the program. In particular, he could discover whether or not it was encrypted with the right key k. Our solution makes decrypt total: in case of wrong key y will be bound to $\bot$ but the last message will be sent anyway. We leave to the programmer the task of handling failures.

To guarantee a safe use of cryptography we also assume that every expression Exp different from the five above and every boolean expression B different from the equality test will fail when applied to ciphertexts, producing a $\bot$. This is important to (i) avoid "magic" expressions which decrypt a ciphertext without knowing the key like, e.g., $\mathsf{magicdecrypt}(\{v, c\}_n) = v$; (ii) abstract away from the bit-stream representation of ciphertexts: in our model, doing any kind of arithmetic operation on a ciphertext has an unpredictable result given that we are assuming randomized encryption. Checking equality is instead useful to observe copies of the very same encryption.

3. Standard noninterference

In this Section, we recall the notion of *strongly secure* programs [15, 12] and we naively try to apply it to our setting so to illustrate why it does not scale to cryptography. In doing this, we will exploit arguments similar to the ones of [5].

Strongly Secure Programs This notion of NI is based on an underlying relation $=_L$ equating states which are indistinguishable by low level users. The intuition is that low level users can observe every low level variable $l \in L$ and every network channel $cid \in CID$. We let a *state* $s = (m, \sigma)$ be a pair composed of a memory m and a network state σ.

Definition 3.1. *Two states $s_1 = (m_1, \sigma_1)$ and $s_2 = (m_2, \sigma_2)$ are* low-equivalent*, noted $s_1 =_L s_2$, if*

1. $m_1 =_L m_2$*:* $\forall l \in L, m_1(l) = m_2(l)$*;*
2. $\sigma_1 =_L \sigma_2$*:* $\forall cid \in CID, \sigma_1(cid) = \sigma_2(cid)$*;*

In order to judge a multi-threaded program $\vec{C}$ secure we employ the notion of *strong low-bisimilarity* [15]: two strongly low-bisimilar thread pools must be of the same size and must spawn or terminate the same number of threads at each execution step, moreover each sequential move of one thread pool must be simulated by corresponding thread of the low-bisimilar pool and lead from low-equivalent states into low-equivalent states. Formally:

Definition 3.2. *Let $\mathcal{R} \subseteq \bigcup_{n \in \mathbb{N}} (CMD^n \times CMD^n)$ be a symmetric relation on multi-threaded programs of equal size. $\mathcal{R}$ is a* strong low-bisimulation *if whenever*

$$\langle C_1 \ldots C_n \rangle \mathcal{R} \langle D_1 \ldots D_n \rangle$$

then $\forall s_1, s_2, i$, such that $s_1 =_L s_2$:

$$\langle\!\langle\, C_i, s_1 \,\rangle\!\rangle \rightarrow \langle\!\langle\, \vec{C'}, s_1' \,\rangle\!\rangle \text{ implies } \langle\!\langle\, D_i, s_2 \,\rangle\!\rangle \twoheadrightarrow \langle\!\langle\, \vec{D'}, s_2' \,\rangle\!\rangle$$
for some $\vec{D'}, s_2'$ such that $\vec{C'} \mathcal{R} \vec{D'}, s_1' =_L s_2'$.

Strong low-bisimilarity $\approxeq_L$ *is defined as the union of all strong low-bisimulations.*

Table 1 Semantics

Commands

$$[\textit{skip}] \quad \langle\!|\, \mathsf{skip}, m, \sigma \,|\!\rangle \twoheadrightarrow \langle\!|\, \langle\rangle, m, \sigma \,|\!\rangle$$

$$[\textit{assign}] \quad \frac{Exp \downarrow^m v}{\langle\!|\, var := Exp, m, \sigma \,|\!\rangle \twoheadrightarrow \langle\!|\, \langle\rangle, m[var \mapsto v], \sigma \,|\!\rangle}$$

$$[\textit{seq1}] \quad \frac{\langle\!|\, C_1, m_1, \sigma \,|\!\rangle \twoheadrightarrow \langle\!|\, \langle\rangle, m_2, \sigma' \,|\!\rangle}{\langle\!|\, C_1; C_2, m_1, \sigma \,|\!\rangle \twoheadrightarrow \langle\!|\, C_2, m_2, \sigma' \,|\!\rangle}$$

$$[\textit{seq2}] \quad \frac{\langle\!|\, C_1, m_1, \sigma \,|\!\rangle \twoheadrightarrow \langle\!|\, C_1'\vec{D}, m_2, \sigma' \,|\!\rangle}{\langle\!|\, C_1; C_2, m_1, \sigma \,|\!\rangle \twoheadrightarrow \langle\!|\, (C_1'; C_2)\vec{D}, m_2, \sigma' \,|\!\rangle}$$

$$[\textit{ift}] \quad \frac{B \downarrow^m \mathsf{true}}{\langle\!|\, \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, m, \sigma \,|\!\rangle \twoheadrightarrow \langle\!|\, C_1, m, \sigma \,|\!\rangle}$$

$$[\textit{iff}] \quad \frac{B \downarrow^m \mathsf{false} \ \vee \ B \downarrow^m \bot}{\langle\!|\, \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, m, \sigma \,|\!\rangle \twoheadrightarrow \langle\!|\, C_2, m, \sigma \,|\!\rangle}$$

$$[\textit{whilet}] \quad \frac{B \downarrow^m \mathsf{true}}{\langle\!|\, \mathsf{while}\ B\ \mathsf{do}\ C, m, \sigma \,|\!\rangle \twoheadrightarrow \langle\!|\, C; \mathsf{while}\ B\ \mathsf{do}\ C, m, \sigma \,|\!\rangle}$$

$$[\textit{whilef}] \quad \frac{B \downarrow^m \mathsf{false} \ \vee \ B \downarrow^m \bot}{\langle\!|\, \mathsf{while}\ B\ \mathsf{do}\ C, m, \sigma \,|\!\rangle \twoheadrightarrow \langle\!|\, \langle\rangle, m, \sigma \,|\!\rangle}$$

$$[\textit{fork}] \quad \langle\!|\, \mathsf{fork}(C\vec{D}), m, \sigma \,|\!\rangle \twoheadrightarrow \langle\!|\, C\vec{D}, m, \sigma \,|\!\rangle$$

$$[\textit{send}] \quad \frac{Exp \downarrow^m v \quad \sigma(cid) \ = \ vals}{\langle\!|\, \mathsf{send}(cid, exp), m, \sigma \,|\!\rangle \twoheadrightarrow \langle\!|\, \langle\rangle, m, \sigma[cid \mapsto v.vals] \,|\!\rangle}$$

$$[\textit{receive}] \quad \frac{\sigma(cid) \ = \ vals.v}{\langle\!|\, \mathsf{receive}(cid, var), m, \sigma \,|\!\rangle \twoheadrightarrow \langle\!|\, \langle\rangle, m[var \mapsto v], \sigma[cid \mapsto vals] \,|\!\rangle}$$

Threads

$$[\textit{proc}] \quad \frac{\langle\!|\, C_i, m_1, \sigma \,|\!\rangle \twoheadrightarrow \langle\!|\, \vec{C}, m_2, \sigma' \,|\!\rangle}{\langle\!|\, \langle C_1 \ldots C_i \ldots C_n\rangle, m_1, \sigma \,|\!\rangle \rightarrow \langle\!|\, \langle C_1 \ldots \vec{C} \ldots C_n\rangle, m_2, \sigma' \,|\!\rangle}$$

Distributed programs

$$[\textit{par}] \quad \frac{\langle\!|\, \vec{C}_j, m_j, \ \sigma \,|\!\rangle \rightarrow \langle\!|\, \vec{C}_j', m_j', \ \sigma' \,|\!\rangle}{\lhd(\vec{C}_1, m_1) \ldots (\vec{C}_n, m_n); \sigma\rhd \twoheadrightarrow \lhd(\vec{C}_1, m_1) \ldots (\vec{C}_j', m_j') \ldots (\vec{C}_n, m_n); \sigma'\rhd}$$

The intuition behind $\vec{C} \approx_L \vec{D}$ is that the two programs $\vec{C}$ and $\vec{D}$ are not distinguishable by any low level observer. In fact, every change done by one computational step of $\vec{C}$ to the state is simulated by $\vec{D}$ in a way that preserves state low-equivalence. Thus, if the states were indistinguishable before such a step, they will remain indistinguishable even after. Moreover, the reached programs have to be bisimilar so to guarantee that even the future steps will be indistinguishable. Notice also the universal quantification over all the possible low-equivalent states done at each step. This ensures compositionality given that any state change possibly performed by parallel threads or distributed programs is certainly "covered" by quantifying over all the possible states.

Definition 3.3. *A program $\vec{C}$ is* secure *if $\vec{C} \approx_L \vec{C}$.*

Intuitively, a secure program which is run on equivalent states will always produce low-equivalent states, even in the presence of parallel threads and distributed programs. Thus no information on the high variables will ever be leaked.

Standard NI and Cryptography The above noninterference notion is too restrictive if naively applied to our language with cryptography. A simple assignment

$$l := \mathsf{encrypt}(h, k) \tag{2}$$

of an encrypted high level value to a low variable would be considered insecure. Notice that this kind of assignments are the one we expect to be able to do via encryption: we actually want to hide high level information via cryptography so to, e.g., safely send it on the untrusted network or simply store it in an untrusted (low) part of the local memory. To see why this simple program is judged to be insecure, consider the two following low-equivalent memories $m_1 =_L m_2$:

m_1	m_2
$h : 1234$	$h : 5678$
$l : 0$	$l : 0$
$k : K$	$k : K$

Running the assignment we get

m_1'	m_2'
$h : 1234$	$h : 5678$
$l : \{1234, c_1\}_K$	$l : \{5678, c_2\}_K$
$k : K$	$k : K$

where $m_1'(l) = \{1234, c_1\}_K \neq \{5678, c_2\}_K = m_2'(l)$ and so $m_1' \neq_L m_2'$. We conclude that

$$l := \mathsf{encrypt}(h, k) \not\approx_L l := \mathsf{encrypt}(h, k)$$

and, consequently, such an assignment is judged as insecure. Notice that this is not just caused by the counfounders c_1 and c_2. Even without confounders the two ciphertexts $\{1234\}_k$ and $\{5678\}_k$ would differ. The problem is related to the fact that $=_L$ do no take into account that the plaintext should not be visible without knowing the decryption key. Counfounders will actually help us defining a new notion of low-equivalence which is suitable for cryptography.

4. Cryptographic noninterference

In this Section, we adapt the notion of *strongly secure programs* illustrated in the previous Section to our language with cryptography. Interestingly, we will only refine the underlying notion of *low-equivalent* states $=_L$ leaving the remaining part of the definition, i.e., the low-bisimulation part, substantially untouched. As we will show in Section 6, this minimal change together with the fact that cryptography is modeled via expressions, thus leaving the language unchanged, will make it very easy to rephrase an existing type system to the new setting.

Low-equivalent ciphertexts The use of confounders allows us to assume that encryptions will always be different, even when the encrypted messages are the same. As already discussed, this is an abstraction of randomized encryption, where encryptions are not always different but the probability that they are the same is negligible.

Intuitively, if ciphertexts are always different we can consider them to be indistinguishable and so equate them all. Of course, this is not true for values encrypted with a low level key, i.e., a key known by low level users. We thus need to distinguish high level keys from low level ones. One solution could be to reuse the partitioning of variables into $\mathcal{H}$ and $\mathcal{L}$ and ask that high level keys are stored into high variables. However this choice is not adequate given that we want to allow flows from $\mathcal{L}$ to $\mathcal{H}$, e.g., $h := l$, and we would not be guaranteed that performing an encryption with a high variable h, e.g., $\mathsf{encrypt}(h', h)$, would result in a ciphertext that cannot be read by the environment.

We thus assume that variables VAR are partitioned into three disjoint sets $\mathcal{H}$, $\mathcal{L}$ and $\mathcal{K}$, where $\mathcal{H}$ and $\mathcal{L}$ are as before and $\mathcal{K}$, ranged over by k, is the set of variables containing *high level keys*. High and low variables, when used as keys, will represent *low level keys*, given that low level values are allowed to flow into high level variables, as explained above. We let high level key values K range over KEY, the subset of the atomic values used as high level keys.

We give a new version of low-equivalence on values called cryptographic-low-equivalence, noted $\approx_C$, and based on the notion of patterns of [3, 2]. We extend the set of possible values with $\Box_c$, representing an undecryptable message with confounder c and we call *patterns* (PAT) this set of extended values. We then define a function $\mathsf{p}(v)$ which takes a value and returns the corresponding pattern by replacing with $\Box_c$ all the ciphertexts that cannot be decrypted. The intuition is that equal confounders correspond to equal, undecryptable, messages. In fact, messages generated using high keys are always assumed to contain a fresh confounder making it impossible to have two different messages with the same confounder. Given a bijection on confounders $\rho : \mathcal{C} \to \mathcal{C}$, that we call *confounder substitution*, we write $p\rho$ to denote the result of applying ρ as a substitution to the pattern p.

Definition 4.1. *Let $p : VAL \to PAT$ be defined as follows:*

$$\begin{array}{rcll} \mathsf{p}(\bot) & = & \bot & \\ \mathsf{p}(n) & = & n & \\ \mathsf{p}(b) & = & b & \\ \mathsf{p}((v_1, v_2)) & = & (\mathsf{p}(v_1), \mathsf{p}(v_2)) & \\ \mathsf{p}(\{v, c\}_n) & = & \begin{cases} \{\mathsf{p}(v), c\}_n & \text{if } n \notin KEY \\ \Box_c & \text{otherwise} \end{cases} & \end{array}$$

Two values v_1 and v_2 are cryptographically-low-equivalent, written $v_1 \approx_C v_2$, if there exists a confounder substitution ρ such that $\mathsf{p}(v_1) = \mathsf{p}(v_2)\rho$.

Let us see $\approx_C$ at work through some simple examples. Consider again Program (2) of Section 3 which, starting from low-equivalent memories, was producing the two different ciphertexts $\{1234, c_1\}_K$ and $\{5678, c_2\}_K$. Given that $K \in KEY$, we obtain that $\mathsf{p}(\{1234, c_1\}_K) = \Box_{c_1}$ and $\mathsf{p}(\{5678, c_2\}_K) = \Box_{c_2}$. Given that confounders represent random numbers, the two patterns are indistinguishable. In fact, by taking $\rho(c_2) = c_1$ we obtain $\Box_{c_1} = \Box_{c_2}\rho$ and so $\{1234, c_1\}_K \approx_C \{5678, c_2\}_K$; if we change the key into a low level one $n \notin KEY$ we obtain that they are different. In particular, $\mathsf{p}(\{1234, c_1\}_n) = \{1234, c_1\}_n \neq \{5678, c_2\}_n = \mathsf{p}(\{5678, c_2\}_n)$. Notice that it is impossible to make the two patterns equal through a substitution ρ because of the different plaintexts, thus $\{1234, c_1\}_n \not\approx_C \{5678, c_2\}_n$. Notice that the same happens if only one of the two keys is untrusted, e.g., $\{1234, c_1\}_K \not\approx_C \{5678, c_2\}_n$, since $\Box_{c_1}$ is never equal to $\{5678, c_2\}_n$. As a matter of fact, one of the two ciphertexts can be decrypted using n which tells an observer that the first ciphertext is, at least, encrypted with a different key.

By substituting $m_1(l) = m_2(l)$ with $m_1(l) \approx_C m_2(l)$ and $\sigma_1(cid) = \sigma_2(cid)$ with $\sigma_1(cid) \approx_C \sigma_2(cid)$ in Definition 3.1, we could directly obtain a definition of $\approx_C$ for memories, networks and states. However, we will see in a while that $\approx_C$ needs to be applied to the memories and channels as a whole, and not, individually, to each stored value with its corresponding one.

Patterns of equal ciphertexts When we generate new ciphertexts, the counfounder guarantees that they will be dif-

ferent, but if we copy a ciphertext from a variable to another one, that variables will be identical. Intuitively, this correspond to the attacker ability of comparing ciphertexts bit-wise as done, e.g., in traffic-analysis: copies of the same ciphertext will always be identical and we have to consider this aspect when defining low-equivalence. Take, for example, the following low program which only acts on low variables and public channels:

$$\begin{array}{l}\mathsf{if}\ (l_1 = l_2)\ \mathsf{then}\\ \quad \mathsf{send}(cid, l_3)\\ \mathsf{else}\\ \quad \mathsf{send}(cid, l_4)\end{array} \tag{3}$$

Depending on the equality of l_1 and l_2 it sends the value of two different low variables on channel cid. Now, consider the following states:

m_1	σ_1	m_2	σ_2
$l_1 : \{1234, c_1\}_K$	$cid :$	$l_1 : \{9999, c_1'\}_K$	$cid :$
$l_2 : \{1234, c_1\}_K$		$l_2 : \{5678, c_2'\}_K$	
$l_3 : \mathsf{true}$		$l_3 : \mathsf{true}$	
$l_4 : \mathsf{false}$		$l_4 : \mathsf{false}$	

Since $K \in KEY$, we have $m_1(l_1) = \{1234, c_1\}_K \approx_C \{9999, c_1'\}_K = m_2(l_1)$ and $m_1(l_2) = \{1234, c_1\}_K \approx_C \{5678, c_2'\}_K = m_2(l_2)$. However, running the program on these memories, we obtain different network states:

σ_1'	σ_2'
$cid : \mathsf{true}$	$cid : \mathsf{false}$

Thus, m_1 and m_2 should not be considered equivalent. We further illustrate this crucial issue with another simple example. Consider the two programs:

$l_1 := \mathsf{encrypt}(h, k)$	$l_1 := \mathsf{encrypt}(h, k)$
$l_2 := \mathsf{encrypt}(h, k)$	$l_2 := l_1$

Starting from clearly low-equivalent memories, in which, e.g., $l_1 = l_2 = 0$, they will produce the following memories

m_1	m_2
$h : 1234$	$h : 5678$
$l_1 : \{1234, c_1\}_K$	$l_1 : \{5678, c_3\}_K$
$l_2 : \{1234, c_2\}_K$	$l_2 : \{5678, c_3\}_K$
$k : K$	$k : K$

(4)

Notice that $m_1(l_1) \approx_C m_2(l_1)$ and $m_1(l_2) \approx_C m_2(l_2)$ but in m_1 we have $l_1 \neq l_2$ while in m_2 we have $l_1 = l_2$. Thus the previously discussed low Program (3) would distinguish m_1 from m_2 even if they are considered low-equivalent by $\approx_C$.

One might argue that the two programs above are anyway distinguishable by NI, given that we quantify over all possible low-equivalent states. After the first assignment to l_1 we might, in fact, override that value and take two equivalent memories with, e.g, $l_1 = 0$. If we run the programs on these new memories we clearly obtain non-equivalent memories, given that we will have $l_2 : \{1234, c_2\}_K$ for the first program and $l_2 : 0$ for the second one. NI can thus track copies between memory cells via plain-texts and given that we require bisimilar programs to preserve low-equivalence of all the possible low-equivalent states, the two programs above would result to be non-bisimilar. However we may write a smarter program that copies l_1 to l_2 only when l_1 actually contains a ciphertext:

$$\begin{array}{l}\mathsf{if}\ (\mathsf{decrypt}(l_1, k) \neq \bot)\ \mathsf{then}\\ \quad l_2 := l_1\\ \mathsf{else}\\ \quad l_2 := \mathsf{encrypt}(h, k)\end{array}$$

This program produces memories like (4) only when l_1 is actually a ciphertext encrypted with k, but we cannot track anymore this copy via plain-texts because, when l_1 is not a ciphertext, the program writes a new fresh ciphertext to l_2 which will never be the same as l_1. Notice that, whenever we observe $l_1 = l_2$, e.g., via the low Program (3), we learn that l_1 is encrypted with K, which should be detected as a flow from high to low.

These examples show that we need to build one single pattern on the whole memory, so that equal confounders in different memory cells will be observable. Indeed, the same reasoning applies to the network: equal confounders appearing either in the local memory or on the network channels will be observable, too. In order to deal with channel values, we extend patterns to deal with list of values, noted $v_1.vals$, by just letting $\mathsf{p}(v_1.vals) = \mathsf{p}(v_1).\mathsf{p}(vals)$. As before, we have that $vals \approx_C vals'$ if $\mathsf{p}(vals) = \mathsf{p}(vals')\rho$ for a confounder substitution ρ.

Definition 4.2. *The set of memory, network and state* patterns *are constructed as follows:*

$$\begin{array}{rcl}\mathsf{sp}(m) & = & \{\ (\ l,\ \mathsf{p}(m(l))\ )\ |\ \forall l \in L\ \}\\ \mathsf{sp}(\sigma) & = & \{\ (\ cid,\ \mathsf{p}(\sigma(cid))\ )\ |\ \forall cid \in CID\ \}\\ \mathsf{sp}(m, \sigma) & = & \mathsf{sp}(m) \cup \mathsf{sp}(\sigma)\end{array}$$

Two memories, networks or states t_1 and t_2 are cryptographically-low-equivalent*, written $t_1 =_C t_2$, if there exists a confounder substitution ρ such that $\mathsf{sp}(t_1) = \mathsf{sp}(t_2)\rho$.*

For example, memories (4) have state patterns

$$\mathsf{sp}(m_1) = \{\ (l_1, \square_{c_1}),\ (l_2, \square_{c_2})\ \}$$

and

$$\mathsf{sp}(m_2) = \{\ (l_1, \square_{c_3}),\ (l_2, \square_{c_3})\ \}$$

Notice that there not exists a substitution ρ which makes such state patterns equal. We thus conclude that $m_1 \neq_C m_2$.

This reflects the fact that equality of confounders is not the same in the two states: if equality is not preserved, it is

in fact impossible to find a bijection ρ on confounders that make them the same.

We can prove that equivalent states are such that values of corresponding variables and channels are equivalent, too. The opposite implication does not hold and it actually motivated the definition of state patterns. We can also prove that removing (i.e., reading) the first value of one channel (item 3) and also copying it to the same low variable (item 4) do not break state equivalence.

Lemma 4.3. *If* $(m_1, \sigma_1) =_C (m_2, \sigma_2)$ *then*

1. $m_1 =_C m_2$ *also implying* $\forall l, m_1(l) \approx_C m_2(l)$*;*
2. $\sigma_1 =_C \sigma_2$ *also implying* $\forall cid, \sigma_1(cid) \approx_C \sigma_2(cid)$*;*
3. *If* $\sigma_1(cid) = vals_1.v_1$ *and* $\sigma_2(cid) = vals_2.v_2$ *then* $(m_1, \sigma_1[cid \mapsto vals_1]) =_C (m_2, \sigma_2[cid \mapsto vals_2])$.
4. *If* $\sigma_1(cid) = vals_1.v_1$ *and* $\sigma_2(cid) = vals_2.v_2$ *then* $(m_1[l \mapsto v_1], \sigma_1[cid \mapsto vals_1]) =_C (m_2[l \mapsto v_2], \sigma_2[cid \mapsto vals_2])$.

Proof. The first two statements derives from the fact that we take subsets of the state patterns. On those subsets we can apply the same ρ that we used to equate states. For example, $(m_1, \sigma_1) =_C (m_2, \sigma_2)$ if $\mathsf{sp}(m_1, \sigma_1) = \mathsf{sp}(m_2, \sigma_2)\rho$. We have that $\mathsf{sp}(m_1)$ and $\mathsf{sp}(m_2)$ are the subsets of $\mathsf{sp}(m_1, \sigma_1)$ and $\mathsf{sp}(m_2, \sigma_2)$ only containing variables with their patterns. It is thus easy to see that the same ρ equates such memory patterns, i.e., $\mathsf{sp}(m_1) = \mathsf{sp}(m_2)\rho$. Analogously for statement three and four: removing the first value of a channel leaves the remaining patterns identical, up to ρ; the same happens when we assign such value to a low variable. □

The next last example, taken from [5], shows why it is important to simultaneously observe patterns of memories and channels, as we do.

$$
\begin{array}{l}
l := \mathsf{encrypt}(h_1, k) \\
\mathsf{send}(ch, l) \\
\mathsf{if}\ h\ \mathsf{then} \\
\quad l := \mathsf{encrypt}(h_2, k) \\
\mathsf{else} \\
\quad \mathsf{skip}
\end{array}
\tag{5}
$$

An observer can deduce the value of h by comparing the value of l with the one sent on ch: they will be different only when h is true. Consider the following states just before the branch:

m_1	σ_1
$h : \mathsf{true}$	
$h_1 : 1234 \;\; h_2 : 5678$	
$l : \{1234, c_1\}_K$	$ch : \{1234, c_1\}_K$

m_2	σ_2
$h : \mathsf{false}$	
$h_1 : 4443 \;\; h_2 : 5556$	
$l : \{4443, c_1'\}_K$	$ch : \{4443, c_1'\}_K$

After the branch, m_1 is updated (yielding m_1') with the new value $\{5678, c_2\}_K$ for variable l while m_2 will not be touched ($m_2 = m_2'$). If we only observe memory patterns we have $\mathsf{sp}(m_1') = \{\ (l, \square_{c_2})\ \} =_C \{\ (l, \square_{c_1'})\ \} = \mathsf{sp}(m_2')$, since they are equal, up to renaming of c_1' into c_2. This is because there are no copies of l in the memory to compare with. Similarly we have $\mathsf{sp}(\sigma_1) = \mathsf{sp}(\sigma_2)$. However, if we observe the whole state we obtain $\mathsf{sp}(m_1', \sigma_1) = \{\ (l, \square_{c_2}),\ (ch, \square_{c_1})\ \} \neq_C \{\ (l, \square_{c_1'}),\ (ch, \square_{c_1'})\ \} = \mathsf{sp}(m_2', \sigma_2)$. Notice that the equality of c_1' in $\mathsf{sp}(m_2', \sigma_2)$ makes it impossible to rename confounders so to make patterns equal. Intuitively, the comparison with the value sent on the network allows us to deduce the value of h.

Secure programs We define *Strong cryptographic low-bisimilarity*, noted $\cong_C$, exactly as strong low-bisimilarity of Definition 3.2, with $=_L$ replaced by $=_C$. When quantifying over all the possible states, we make two assumptions:

Confounder unicity Values encrypted with high level keys and with the same confounder are exactly the same. As already discussed, this is a consequence of using a fresh confounder for each encryption. Instead, we do not assume anything on confounders that might have been chosen by the intruder, i.e., the confounders of ciphertexts encrypted with low level keys;

High level key safety High key variables $k \in \mathcal{K}$ can only contain high key values $K \in KEY$. On the other hand, we never allow high key values to occur unprotected in the low level memory and on the network, given that this would imply those keys are broken. This does not mean we assume keys cannot be broken: since we start from a state with no broken key and we require that, at each steps, keys are not broken, we basically check that high key values remain protected. Moreover, we will prove that this check is actually not necessary because secure programs will never break keys.

First assumption is just a well-formedness condition that is preserved by each program execution, independently of its security:

Definition 4.4. *A state* $s = (m, \sigma)$ *is* well-formed *if whenever* $\{v, c\}_K, \{v', c'\}_{K'}$ *occur in* s*, with* $K, K' \in KEY$*, then* $c = c'$ *implies* $\{v, c\}_K = \{v', c'\}_{K'}$.

We will always implicitly assume that states are well-formed. The second condition, instead, is important to check that high level keys are safely dealt with. In order to formalize it, given a state s, we write $s \vdash k$ to denote that $k \in values(\mathsf{sp}(s))$ where $values(p)$ is the set of all atomic values occurring in pattern p.

Definition 4.5. *A state* $s = (m, \sigma)$ *is* key-safe *if*

1. $\forall k \in \mathcal{K}$, $m(k) \in KEY$*;*
2. $s \vdash n$ *implies* $n \notin KEY$*;*

We denote with KS *the set of key-safe states.*

We are now ready to give the new notion of strong cryptographic low-bisimilarity. Notice that we require key-safety only for quantified states s_1 and s_2. Indeed, we will prove that key-safety is preserved by bisimilar programs with controlled assignments to high level variables.

Definition 4.6. *Let $\mathcal{R} \subseteq \bigcup_{n\in\mathbb{N}} (CMD^n \times CMD^n)$ be a symmetric relation on multi-threaded programs of equal size. $\mathcal{R}$ is a* strong cryptographic low-bisimulation *if whenever $\langle C_1 \dots C_n\rangle \mathcal{R} \langle D_1 \dots D_n\rangle$ then $\forall i$, $\forall s_1, s_2 \in KS$, such that $s_1 =_C s_2$:*

$$\langle\!\langle\, C_i, s_1 \,\rangle\!\rangle \twoheadrightarrow \langle\!\langle\, \vec{C}', s_1' \,\rangle\!\rangle \text{ implies } \langle\!\langle\, D_i, s_2 \,\rangle\!\rangle \twoheadrightarrow \langle\!\langle\, \vec{D}', s_2' \,\rangle\!\rangle$$
for some $\vec{D}', s_2'$ such that $\vec{C}' \mathcal{R} \vec{D}', s_1' =_C s_2'$.

Strong cryptographic low-bisimilarity $\approxeq_C$ *is defined as the union of all strong cryptographic low-bisimulations.*

The universal quantification over all possible cryptographically–low–equivalent states done at each step ensures compositionality. Indeed, any state change performed by a (possibly evil) concurrent thread or distributed program will be certainly "covered" by this quantification.

We can now prove key-safety preservation for programs with controlled assignments to high level key variables, called *key-safe programs*:

Definition 4.7. *A program C is key-safe if*

1. receive(cid, k) *never occurs in C;*
2. k:=*Exp occurring in C implies Exp $= k'$ or Exp $=$* newkey.

Proposition 4.8. *Let C and D be two key-safe programs. If $\forall s_1, s_2 \in KS$, with $s_1 =_C s_2$,*

$$\langle\!\langle\, C, s_1 \,\rangle\!\rangle \twoheadrightarrow \langle\!\langle\, \vec{C}', s_1' \,\rangle\!\rangle \text{ implies } \langle\!\langle\, D, s_2 \,\rangle\!\rangle \twoheadrightarrow \langle\!\langle\, \vec{D}', s_2' \,\rangle\!\rangle$$
for some $\vec{D}', s_2'$ such that $s_1' =_C s_2'$

then $s_1', s_2' \in KS$.

Proof. Let us assume, by contradiction, that one of s_1', s_2', let us say s_1', is not in KS. We have that either $m_1'(k) \notin KEY$ for a certain k, or $s_1' \vdash K$ with $K \in KEY$. Since $s_1 \in KS$ we have that $\forall k', m_1(k') \in KEY$. The assumption on assignments ensures that k can only have been assigned to another k', for which we know $m_1(k') \in KEY$, or to newkey that, by definition, returns a value in KEY which gives a contradiction. The only remaining case is $s_1' \vdash K$ with $K \in KEY$. We have two sub-cases: (i) if K has been generated with newkey it must be different from every other name in s_2' and, since it appears in $\mathsf{sp}(s_1')$ and not in $\mathsf{sp}(s_2')$, it cannot be $s_1' =_C s_2'$; (ii) K appeared in s_1 but not in $\mathsf{sp}(s_1)$ because $s_1 \in KS$. Thus we can consider a new state $s_3 = s_1\eta$ with η being the substitution $K \mapsto K'$, with $K' \leftarrow KEY$ fresh. Since $\mathsf{sp}(s_1) = \mathsf{sp}(s_3)$, then we have $s_1 =_C s_3$. We can run again C and D on equivalent states s_1 and s_3. Since K does not occur in s_3 it is impossible that it appears in s_3'. But we have that K appeared in $\mathsf{sp}(s_1')$ from which we obtain the contradiction, i.e., $s_1' \neq_C s_3'$. Intuitively, the universal quantification on equivalent states allows us to relabel broken key K to a fresh one and observe its leakage by comparison with the relabelled state (which does not contain it). Assumption on receive() command let us preserve key-safe while reading messages from the network. This follow directly form the fact that $s_1, s_2 \in KS$, so $s_1 \nvdash K$ and $s_2 \nvdash K$, $\forall K \in KEY$. Thus, we avoid to assign a bad value (i.e., a valued not contained in KEY) to a key. □

Definition 4.9. *A program $\vec{C}$ is* secure *if it is key-safe and $\vec{C} \approxeq_C \vec{C}$.*

5. Hook-up properties

Inspired by [15, 12], in this section we investigate hook-up properties for $\approxeq_C$, i.e., compositionality results among secure programs. The results we prove are very similar to the ones of [15, 12], but we need to carefully adapt the notion of *low* expressions. A *low* expression, in the model with no cryptography, is just an expression that evaluates the same when calculated on low-equivalent states [15, 12], i.e.,

$$\forall\, m_1 =_L m_2,\ \ Exp \downarrow^{m_1} = Exp \downarrow^{m_2}$$

We cannot just rephrase it with the new equivalences

$$\forall\, m_1 =_C m_2,\ \ Exp \downarrow^{m_1} \approx_C Exp \downarrow^{m_2} \tag{6}$$

The problem is that $Exp \downarrow^{m_1} \approx_C Exp \downarrow^{m_2}$ does not guarantee that if such values are stored in the memories or sent on public channels the resulting state will be equivalent. In particular, it does not hold that $m_1 =_C m_2$ and $Exp \downarrow^{m_1} \approx_C Exp \downarrow^{m_2}$ imply $m_1[l \mapsto Exp \downarrow^{m_1}] \approx_C m_2[l \mapsto Exp \downarrow^{m_2}]$, and analogously for network states, as we have already illustrated in example (**??**). We thus give the following stronger requirement for low expressions:

Definition 5.1. *An expression Exp is said to be* low *if, $\forall l \in L, \forall cid \in CID$, for all states $(m_1, \sigma_1) =_C (m_2, \sigma_2)$, if we let $v_1 = Exp \downarrow^{m_1}$ and $v_2 = Exp \downarrow^{m_2}$, it holds that*

1. $(m_1[l \mapsto v_1], \sigma_1) =_C (m_2[l \mapsto v_2], \sigma_2)$;
2. $(m_1, \sigma_1[cid \mapsto v_1.vals]) =_C (m_2, \sigma_2[cid \mapsto v_2.vals])$.

otherwise Exp is high.

We have the following simple lemma, stating that the previously proposed definition of low expressions (5) is implied by our new definition.

Lemma 5.2. *If Exp is* low *then $\forall m_1 =_C m_2$, we have $Exp \downarrow^{m_1} \approx_C Exp \downarrow^{m_2}$.*

Proof. This fact is a direct consequence of Lemma 4.3, item 1. Taking a σ with empty channels, we also have that $(m_1, \sigma) =_C (m_2, \sigma)$. By definition of low expressions, we obtain $(m_1[l \mapsto Exp \downarrow^{m_1}], \sigma) =_C (m_2[l \mapsto Exp \downarrow^{m_2}], \sigma)$ and, by Lemma 4.3, item 1, $Exp \downarrow^{m_1} \approx_C Exp \downarrow^{m_2}$. □

This lemma is useful for proving a deterministic behaviour in case of low boolean guards. Notice the equality instead of equivalence:

Corollary 5.3. *If B is low, then $B \downarrow^{m_1} = B \downarrow^{m_2}$.*

Proof. Trivial, by Lemma 5.2 and by the fact that $\mathsf{p}(b) = b$ (Definition 4.1). □

We extend the definition of secure contexts of [15, 12] by taking into account direct assignment to high level key variables and a careful use of the receive command.

Definition 5.4. *A* secure context *is a context built with secure programs. Let $[\vec{\bullet}]$ and $[\bullet]$ be holes for, respectively, a command vector and singleton command. Secure contexts are defined as follows*

$$\begin{aligned}\mathbb{C}[\vec{\bullet_1}, \vec{\bullet_2}] ::= {} & \mathsf{skip} \mid h := Exp \mid l := Exp_L \mid [\bullet_1]; [\bullet_2] \\ & \mid k := k' \ (k, k' \in \mathcal{K}) \mid k := \mathsf{newkey} \ (k \in \mathcal{K}) \\ & \mid \mathsf{if}\, B_L \,\mathsf{then}\, [\bullet_1] \,\mathsf{else}\, [\bullet_2] \mid \mathsf{while}\, B_L \,\mathsf{do}\, [\bullet_1] \\ & \mid \mathsf{fork}([\bullet_1][\vec{\bullet_2}]) \mid \mathsf{send}(cid, Exp_L) \\ & \mid \mathsf{receive}(cid, var) \ (var \notin \mathcal{K}) \mid \langle [\vec{\bullet_1}][\vec{\bullet_2}] \rangle\end{aligned}$$

where B_L and Exp_L denotes low expressions.

The next result proves that $\approx_C$ is preserved by secure contexts.

Theorem 5.5. *If $\vec{C_1} \approx_C \vec{C_1'}$, $\vec{C_2} \approx_C \vec{C_2'}$ then*

1. *$\mathbb{C}[\vec{C_1}, \vec{C_2}] \approx_C \mathbb{C}[\vec{C_1'}, \vec{C_2'}]$;*
2. *Let $\mathbb{D}[\vec{\bullet_1}, \vec{\bullet_2}] = \mathsf{if}\, B \,\mathsf{then}\, \vec{\bullet_1} \,\mathsf{else}\, \vec{\bullet_2}$, with B high. Then, $\vec{C_1} \approx_C \vec{C_2}$ implies $\mathbb{D}[\vec{C_1}, \vec{C_2}] \approx_C \mathbb{D}[\vec{C_1'}, \vec{C_2'}]$.*

Proof outline. (Full proof is in [6]) The Theorem is proved inductively on the structure of contexts, by exploiting equivalences $\vec{C_1} \approx_C \vec{C_1'}$ and $\vec{C_2} \approx_C \vec{C_2'}$ and, for statement 2, $\vec{C_1} \approx_C \vec{C_2}$. It can be conducted as in [15], except for assignments, message exchange, branches and while loops. For assignments and message exchanges, we have to prove that equivalence of states will be preserved by executing the command. To this aim, we directly exploit the requirements on low expression given in Definition 5.1. In fact, such a Definition states that assigning the result of an expression to a low variable or sending such a result on the network leave the states equivalent. For the reception of a message we also exploit Lemma 4.3, item 3 and 4, stating that the removal of a value from a channel and the assignment of that value to a low variable does not break state equivalence. As far as low branches (and while loops) are concerned, we have to prove that they always branch in the same way on equivalent states. This is guaranteed by Corollary 5.3, stating that the result of evaluating B on the two equivalent states is always the same. □

The next Hook-up Corollary proves that secure programs placed in secure contexts are still secure. For high branches, as expected, this happens when the two branches are equivalent.

Corollary 5.6. *Let $\vec{C_1}$, $\vec{C_2}$ be secure programs. Then*

1. *$\mathbb{C}[\vec{C_1}, \vec{C_2}]$ is secure;*
2. *Let $\mathbb{D}[\vec{\bullet_1}, \vec{\bullet_2}] = \mathsf{if}\, B \,\mathsf{then}\, \vec{\bullet_1} \,\mathsf{else}\, \vec{\bullet_2}$, with B high. Then, $\vec{C_1} \approx_C \vec{C_2}$ implies that $\mathbb{D}[\vec{C_1}, \vec{C_2}]$ is secure.*

Proof. Since $\vec{C_1}$ and $\vec{C_2}$ are secure we have $C_1 \approx_C C_1$ and $C_2 \approx_C C_2$. By Theorem 5.5 it must be that $\mathbb{C}[\vec{C_1}, \vec{C_2}] \approx_C \mathbb{C}[\vec{C_1}, \vec{C_2}]$ meaning that $\mathbb{C}[\vec{C_1}, \vec{C_2}]$ is secure, too. □

6. Type system

In this Section, we show that the type system presented by Sabelfeld and Mantel in [12], which is an extension of [15], can be easily adapted to our setting. The type system transforms, if possible, a given program $\vec{C}$ into a new one $\vec{C'}$ which is the timing-leak free version of the original program, by exploiting Agat's approach [4]. In particular, branches of conditional of different lengths are padded using skip commands, when necessary.

The typing rules for commands have the form

$$\vec{C} \hookrightarrow \vec{C'} : \vec{Sl}$$

where $\vec{C}$ is a program, $\vec{C'}$ is its transformation and $\vec{Sl}$ is the type of $\vec{C'}$. The type of a program is its *low slice*: a copy of a secure command where assignment to high and key variables have been replaced by skip. A slice models the time behaviour of $\vec{C'}$ as observable by an attacker running concurrently with it [15].

Our message passing commands send and receive are typed as the low, insecure, channels of [12]. The only real extension to previous type systems are the rules for typing expressions, including encrypt and decrypt.

Expressions Types for expressions are : low, $high$, for low and high data, and key, for high level keys. Only encrypting with a secure key k will provide security guarantees. However, we admit encryption with untrusted (low) values so to allow encrypted communication between the trusted processes and the hostile environment that otherwise could only communicate via plain-texts.

Judgments have the form $Exp : \tau$. Typing rules for expressions are as follows:

$$\text{(var/exp)} \ \ l : low \qquad k : key \qquad Exp : high$$

$$\text{(newkey)} \ \ \mathsf{newkey} : key$$

$$\text{(enc-sec)} \ \ \frac{var : key \quad Exp : high}{\mathsf{encrypt}(Exp, var) : low}$$

$$\text{(op-low)} \ \ \frac{Exp_1 : low, \ldots, Exp_n : low}{op(Exp_1, \ldots, Exp_n) : low}$$

Variables are typed with their respective types, while generic expressions can be always given type $high$. (High variables are typed $high$ being them expressions.) The (newkey) rule states that newkey returns a high level key. The (enc-sec)

Table 2 Typing Commands

$(Skip)$ $\quad \mathsf{skip} \hookrightarrow \mathsf{skip} : \mathsf{skip}$

$(Assign_{low})$ $\quad \dfrac{Exp : \ low}{l := Exp \hookrightarrow l := Exp : l := Exp}$

$(Assign_{high})$ $\quad h := Exp \hookrightarrow h := Exp : \mathsf{skip}$

$(Assign_{key})$ $\quad \dfrac{Exp : \ key}{k := Exp \hookrightarrow k := Exp : \mathsf{skip}}$

(Seq) $\quad \dfrac{C_1 \hookrightarrow C_1' : Sl_1 \ C_2 \hookrightarrow C_2' : Sl_2}{C_1; C_1 \hookrightarrow C_1'; C_2' : Sl_1; Sl_2}$

$(While)$ $\quad \dfrac{B : \ low \ C \hookrightarrow C' : Sl}{\mathsf{while}\ B\ \mathsf{do}\ C \hookrightarrow \mathsf{while}\ B\ \mathsf{do}\ C' : \mathsf{while}\ B\ \mathsf{do}\ Sl}$

$(Fork)$ $\quad \dfrac{C_1 \hookrightarrow C_1' : Sl_1 \ \vec{C_2} \hookrightarrow \vec{C_2'} : \vec{Sl_2}}{\mathsf{fork}(C_1 \vec{C_2}) \hookrightarrow \mathsf{fork}(C_1' \vec{C_2'}) : \mathsf{fork}(Sl_1 \vec{Sl_2})}$

(Par) $\quad \dfrac{C_1 \hookrightarrow C_1' : Sl_1 \ \ldots C_n \hookrightarrow C_n' : Sl_n}{\langle C_1 \ldots C_n \rangle \hookrightarrow \langle C_1' \ldots C_n' \rangle : \langle Sl_1 \ldots Sl_n \rangle}$

(If_{low}) $\quad \dfrac{B : \ low \ C_1 \hookrightarrow C_1' : Sl_1 \ C_2 \hookrightarrow C_2' : Sl_2}{\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \hookrightarrow \mathsf{if}\ B\ \mathsf{then}\ C_1'\ \mathsf{else}\ C_2' : \mathsf{if}\ B\ \mathsf{then}\ Sl_1\ \mathsf{else}\ Sl_2}$

(If_{high}) $\quad \dfrac{B : \ high \ C_1 \hookrightarrow C_1' : Sl_1 \ C_2 \hookrightarrow C_2' : Sl_2 \ al(Sl_1) = al(Sl_2) = \mathsf{false}}{\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \hookrightarrow \mathsf{if}\ B\ \mathsf{then}\ C_1'; Sl_2\ \mathsf{else}\ Sl_1; C_2' : \mathsf{skip}; Sl_1; Sl_2}$

$(Send)$ $\quad \dfrac{Exp : \ low}{\mathsf{send}(cid, Exp) \hookrightarrow \mathsf{send}(cid, Exp) : \mathsf{send}(cid, Exp)}$

$(Receive)$ $\quad \dfrac{var \notin \mathcal{K}}{\mathsf{receive}(cid, var) \hookrightarrow \mathsf{receive}(cid, var) : \mathsf{receive}(cid, \hat{var})}$

rule checks that a proper key is used when a secret text is ciphered: it demands to use a high level key. The (op-low) rule states that expressions whose subexpressions have type low can be typed low. Examples of application of this rule are encryption and decryption with low level keys. Notice that, apart from low decryption, decrypt will be always typed $high$.

Commands Typing and transformation rules are presented in Table 2. Intuitively, the command skip is typed by itself; to prevent explicit flows, rule $(Assign_{low})$ requires the expression to be typed as low; typing an assignment to a secret variable will be done using skip as its low slice, this is because we want that the slice has no occurrences of $high$ variables $(Assign_{high})$. Rule $(Assign_{key})$ requires the expression to be typed as key and uses skip as low slice; Rules $(Seq), (While), (Fork), (Par), (If_{low})$ are as expected and do nothing interesting. Let $al(C)$ be a boolean function on command returning true whenever occurs a syntactic assignment to a low variable or a receive command of the form $\mathsf{receive}(cid, var), var \in \mathcal{L}$. Rule (If_{high}) asks that $(al(Sl_1) = al(Sl_2) = \mathsf{false})$, neither low assignment nor receive that reads message to a low variable occurs, to prevent indirect insecure flows [15]. It also aims to make the two branches of the conditional bisimilar, in fact the transformed command is composed by the same conditional with branches modified to contains the original sub-command and the low slice of the other branch. Rules $(Send)$ and $(Receive)$ are taken from [12] for the case of low (public) channel. We additionally require that a key cannot be read directly from a channel. Low slice cannot use secret variables so $(Receive)$ let the command read the message from the network (removing it from cid) but do not update the variable var if it is high. This is obtained using the notation $\hat{var}$ which is defined as follow: $\hat{h} = \_$, $\hat{l} = l$ [12].

Correctness In order to apply Theorem 5.5 and Corollary 5.6 to well-typed programs, we need the following:

Lemma 6.1. ***(Expression equivalence)*** *If $Exp : low$ then Exp is low according to Definition 5.1*

We finally prove that well-typed programs are secure.

Theorem 6.2. ***(Correctness)***
If $\vec{C} \hookrightarrow \vec{C'} : \vec{Sl}$ then $\vec{C'} \approx_C \vec{Sl}$.

Theorem 6.3. ***(Program Noninterference)***
If $\vec{C} \hookrightarrow \vec{C'} : \vec{Sl}$, then $\vec{C'}$ is secure, i.e., $\vec{C'}$ is key-safe and $\vec{C'} \approx_C \vec{C'}$.

Proof. The proposed type system accepts a program as valid only if it is a key-safe program: rules $(Assign_{key})$ and $(Receive)$ implement the requirements of Definition 4.7. By Theorem 6.2 we know that $\vec{C'} \approx_C \vec{Sl}$. By symmetry and transitivity of $\approx_C$ we get $\vec{C'} \approx_C \vec{C'}$. □

7. Conclusions

We have investigated information flow security for multi-threaded distributed application in the presence of explicit cryptographic operations. The model we have adopted derives from the notion of patterns that has been proposed in [3, 2] for proving computational soundness of formal cryptography. Interestingly, we have adopted it for a completely different purpose, i.e., as an underlying model for rephrasing an existing notion of noninterference [15]. Before discovering we really needed this notion, we have tried a number of different formalizations for low-equivalence, none of them as satisfactory as the present one. As we have discussed in the Introduction, Laud has recently proposed to use the very same notion of patterns as a model for possibilistic noninterference [11]. However, he was aiming at computational soundness results and not at extending the noninterference property, as we have done here. It would be interesting to study in which extent his proof of computational soundness applies to our multi-threaded setting, given the similarity of the models in between noninterference and computational security.

There are many extensions that might be interesting to investigate: keys can be sent around encrypted, but the type system does not allow to use them as keys; we have decided not to approach this issue in order to keep types simple. However, we are confident that this might be easily dealt with by adding some form of channel typing so to safely reconstruct the type of the key upon reception. Public key encryption has not yet been included in the model and could be interesting to investigate how this can be achieved. Finally, we claim that the low-equivalence notion we have proposed is the weakest which is also compositional with respect to low, untyped, programs (representing the opponents). We plan to investigate further this issue and try to prove this claim.

References

[1] Martín Abadi. Secrecy by typing in security protocols. *JACM: Journal of the ACM*, 46, 1999.

[2] Martín Abadi and Jan Jurjens. Formal eavesdropping and its computational interpretation. In *TACS: 4th International Conference on Theoretical Aspects of Computer Software*, 2001.

[3] Martín Abadi and Phillip Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). *JCRYPTOL: Journal of Cryptology*, 15, 2002.

[4] Johan Agat. Transforming out timing leaks. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POLP-00)*, pages 40–53, N.Y., January 19–21 2000. ACM Press.

[5] Aslan Askarov, Daniel Hedin, and Andrei Sabelfeld. Cryptographically-masked flows. In *In Proceedings of the International Static Analysis Symposium*, Seoul, Korea, August 2006. Springer-Verlag.

[6] Matteo Centenaro and Riccardo Focardi. Information flow security of multi-threaded distributed programs (full version). *http://www.dsi.unive.it/~mcentena/centenaro-focardi.pdf*, 2008.

[7] Tom Chothia, Dominic Duggan, and Jan Vitek. Type-based distributed access control. In *In Proceedings of the 16th IEEE Computer Security Foundations Workshop (CSFW '03)*, Asilomar, CA, USA, July 2003.

[8] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7), July 1977.

[9] J. A. Goguen and J. Meseguer. Security policies and security models. In IEEE Computer Society Press, editor, *Proceedings of the 1982 IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[10] Peeter Laud. Secrecy types for a simulatable cryptographic library. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*. ACM SIGSAC, 2005.

[11] Peeter Laud. On the computational soundness of cryptographically masked flows. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 337–348. ACM, 2008.

[12] Andrei Sabelfeld and Heiko Mantel. Static confidentiality enforcement for distributed programs. In *In Proceeding of the 9th International Static Analysis Symposium*, volume 2477, Madrid, Spain, September 2002. Springer.

[13] Andrei Sabelfeld and Heiko Mantel. A unifying approach to the security of distributed and multi-threaded programs. *Journal of Computer Security*, 11(4):615–676, 2003.

[14] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), January 2003.

[15] Andrei Sabelfeld and David Sands. Probabilistic noninterference for multi-threaded programs. In *In Proceedings of the 13th IEEE Computer Security Foundations Workshop*, Cambridge, England, July 2000. IEEE Computer Society.

[16] Geoffrey Smith. A new type system for secure information flow. In *In Proceeding of the 14th IEEE Computer Security Foundations Workshop*, Cape Breton, Nova Scotia, June 2001. IEEE Computer Society.

[17] Geoffrey Smith and Rafael Alpìzar. Secure information flow with random assignment and encryption. In *In Proceeding of the 4th ACM Workshop on Formal Methods in Security Engineering*, Alexandria, Virginia, November 2006. ACM.

[18] Geoffrey Smith and Dennis M. Volpano. Secure information flow in a multi-threaded imperative language. In *In Proceeding of the 25th ACM Symposium on Principles of Programming Languages*, San Diego, California, January 1998.

[19] Jeffrey A. Vaughan and Steve Zdancewic. A cryptographic decentralized label model. In *IEEE Symposium on Security and Privacy*, 2007.