Secure Information Flow for Distributed Systems

Rafael Alpízar and Geoffrey Smith

School of Computing and Information Sciences, Florida International University,
Miami, FL 33199, USA
{ralpi001,smithg}@cis.fiu.edu

Abstract. We present an abstract language for distributed systems of processes with local memory and private communication channels. Communication between processes is done via messaging. The language has high and low data and is limited only by the Denning restrictions; this is a significant relaxation as compared to previous languages for concurrency. We argue that distributed systems in the abstract language are *observationally deterministic*, and use this result to show that well-typed systems satisfy termination-insensitive noninterference; our proof is based on concepts of *stripping* and *fast simulation*, which are a valuable alternative to *bisimulation*. We then informally explore approaches to implement this language concretely, in the context of a wireless network where there is a risk of eavesdropping of network messages. We consider how asymmetric cryptography could be used to realize the confidentiality of the abstract language.

1 Introduction

In this paper we craft a high-level imperative language for distributed systems. Our goal is to provide the programmer with a simple and safe abstract language, with a built-in API to handle communications between processes. The abstract language should hide all messy communication protocols that control the data exchange between processes and all the cryptographic operations that ensure the confidentiality of the data transmitted. We also want to classify variables into different security levels, and we want a secure information flow property that says that distributed system cannot leak information from higher to lower levels. We would like our language to have a clean familiar syntax and to have the maximum power of expression that we can give it.

A distributed system thus, is a group of programs executing in a group of nodes such that there is at least one program per node. An executing program with its local data is a process. As our processes may reside in separate nodes, they should have their own private memories and be able to send and receive messages from other processes.

We would like to classify data according to a security lattice, which in our case will be limited to H and L, and we would like to maintain the ability to transmit and receive H and L values. To do this we will need separate channels for each classification, otherwise we would only be able to receive messages using H variables as demonstrated in adversary Δ_1 of Figure 1. In this attack, Process 1 sends a H variable on channel a, but Process 2 receives it into a L variable. Because of subsumption, H channels can transmit H or L data but the receiving variable must be typed H while L channels can only transmit L data. Therefore our communication channels must have a security

P. Degano and J. Guttman (Eds.): FAST 2009, LNCS 5983, pp. 126–140, 2010.

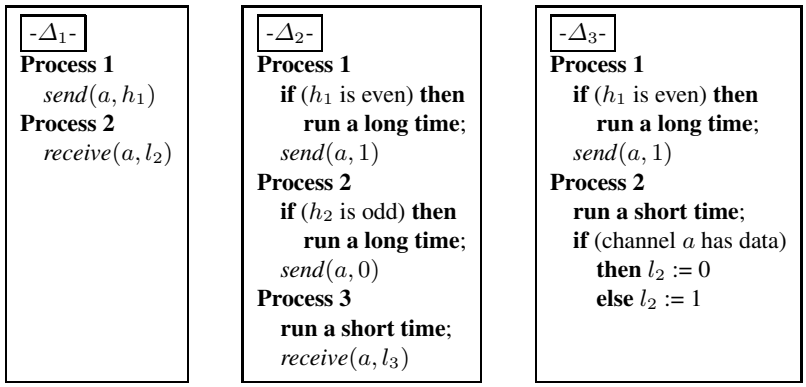


Fig. 1. Attacks (first wave)

classification. What else do they need? We shall see that channels also need a specific source process and a specific destination process. The reason is exemplified by distributed system Δ_2 of Figure 1. In this attack, Process 1 and Process 2 both send on channel a. If we assume that h_1 and h_2 are initialized to the same secret value, then the last bit of this value is leaked into l_3 (assuming sufficiently "fair" scheduling). Our type system prevents attacks Δ_1 and Δ_2 by giving each channel a type of the form $\tau\ ch_{i,j}$ that specifies the security level (τ) and the sending (i) and receiving (j) processes.

Also, processes cannot be allowed to test if a channel has data because this ability would also render the language unsound by allowing timing channels. This is illustrated by distributed system Δ_3 of Figure 1. This attack leaks the last bit of h_1 to l_2. When h_1 is even Process 1 takes a long time to send its message so when Process 2 checks, the channel will be empty. Therefore we do not allow processes to make such tests.

Because a process trying to receive from a channel must block until a message is available, the programmer has to be careful to ensure that for each *receive*, there is a corresponding *send*. The converse, however, is not required since a process may send a message that is never received. Indeed, processes should not be required to wait on *send* and should be able to send multiple times on the same channel. To handle this, we will need an unbounded buffer for each channel, where sent messages wait to be received.

Once we have some idea of what the language should be like, we would like to know that it is safe and argue a noninterference (NI) property on it. But we would like to restrict processes as little as possible. To this end, we explore the possibility of typing processes using only the classic Denning restrictions [1], which disallows an assignment $l := e$ to a L variable if either e contains H variables or if the assignment is within an **if** or **while** command whose guard contains H variables. This would be in sharp contrast to prior works in secure information flow for concurrent programs (such as [2,3,4,5]) which have required severe restrictions to prevent H variables from affecting the *ordering* of assignments to shared memory. Our language, being based on message passing rather than shared memory, is much less dependent on timing and the behavior of the scheduler. Indeed, it turns out that our distributed systems are *observationally deterministic* which means that, despite our use of a purely nondeterministic

process scheduler, the final result of programs is uniquely determined by the initial memories.

Next we would like to explore what it would take to implement our language in a concrete setting. We wish our operational model to be as "close to the ground" as we can. We would like something like a wireless LAN where eavesdroppers can see all communications; what would it take to implement a safe language there? Obviously secret data cannot be transmitted in a wireless LAN with any expectation of confidentiality, hence, we need cryptography. What kind? Asymmetric cryptography seems to be the appropriate style for our setting.

The paper is organized as follows: Section 2 formally defines the abstract language for distributed systems and argues the key noninterference theorem; our proof is not based on bisimulation, however, but instead on concepts of stripping and fast simulation as in [6]. In Section 3 we informally explore what it would take to implement the abstract language in a concrete setting over a public network, and we work out our adversarial model. Section 4 presents related work and Section 5 concludes the paper.

2 An Abstract Language for Distributed Systems

This section defines an abstract language for distributed systems. The language syntax (Figure 2) is that of the simple imperative language except that processes are added to the language and they may send or receive messages from other processes. Note that pseudocommand **done** is used to denote a terminated command. It may not be used as a subcommand of another command, except that for technical reasons we *do* allow the branches of an **if** command to be **done**. Allowing this also has the minor practical benefit of letting us code "**if** e **then** c" by "**if** e **then** c **else done**".

(phrases) $p ::= e \mid c$

(variables) $x, y, z, \ldots$

(channel ids) $a, b, \ldots$

(process ids) $i, j, \ldots$

(expressions) $e ::= x \mid n \mid e_1 + e_2 \mid \ldots$

(commands) $c ::=$ **done** $\mid$ **skip** $\mid x := e \mid$
send$(a, e) \mid$ *receive*$(a, x) \mid$
if e **then** c_1 **else** $c_2 \mid$
while e **do** $c \mid c_1; c_2$

Fig. 2. Abstract Language Syntax

Semantics of processes ($\longrightarrow$): Each process can refer to its local memory μ, which maps variables to integers, and to the global *network memory* Φ, which maps channel identifiers to lists of messages currently waiting to be received; we start execution with an empty network memory Φ_0 such that $\Phi_0(a) = [\,]$, for all a. Thus we specify the semantics of a process via judgments of the form $(c, \mu, \Phi) \longrightarrow (c', \mu', \Phi')$. We use a standard small-step semantics with the addition of rules for the *send* and *receive* commands; the rules are shown in Figure 3. In the rules, we write $\mu(e)$ to denote the value of expression e in memory μ. The rule for *send*(a, e) updates the network memory by adding the value of e to the end of the list of messages waiting on channel a. The rule for *receive*(a, x) requires that there be at least one message waiting to be received on channel a; it removes the first such message and assigns it to x.

Semantics of distributed systems ($\Longrightarrow$): We model a distributed system as a function Δ that maps process identifiers to pairs (c, μ) consisting of a command and a local memory. A *global configuration* then has the form (Δ, Φ), and rule $global_s$ defines the purely nondeterministic behavior of the process scheduler, which at each step can select any process that is able to make a transition.

$$update_s \quad \frac{x \in dom(\mu)}{(x := e, \mu, \Phi) \longrightarrow (\textbf{done}, \mu[x := \mu(e)], \Phi)}$$

$$if_s \quad \frac{\mu(e) \neq 0}{(\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2, \mu, \Phi) \longrightarrow (c_1, \mu, \Phi)} \qquad \frac{\mu(e) = 0}{(\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2, \mu, \Phi) \longrightarrow (c_2, \mu, \Phi)}$$

$$while_s \quad \frac{\mu(e) = 0}{(\textbf{while } e \textbf{ do } c, \mu, \Phi) \longrightarrow (\textbf{done}, \mu, \Phi)} \qquad \frac{\mu_(e) \neq 0}{(\textbf{while } e \textbf{ do } c, \mu, \Phi) \longrightarrow (c; \textbf{while } e \textbf{ do } c, \mu, \Phi)}$$

$$skip_s \quad (\textbf{skip}, \mu, \Phi) \longrightarrow (\textbf{done}, \mu, \Phi)$$

$$compose_s \quad \frac{(c_1, \mu, \Phi) \longrightarrow (\textbf{done}, \mu', \Phi')}{(c_1; c_2, \mu, \Phi) \longrightarrow (c_2, \mu', \Phi')} \qquad \frac{(c_1, \mu, \Phi) \longrightarrow (c_1', \mu', \Phi') \;\; c_1' \neq \textbf{done}}{(c_1; c_2, \mu, \Phi) \longrightarrow (c_1'; c_2, \mu', \Phi')}$$

$$send_s \quad \frac{\Phi(a) = [m_1, \cdots, m_k] \;\; k \geq 0}{(send(a, e), \mu, \Phi) \longrightarrow (\textbf{done}, \mu, \Phi[a := [m_1, \ldots, m_k, \mu(e)]])}$$

$$receive_s \quad \frac{\Phi(a) = [m_1, \ldots, m_k] \;\; k \geq 1}{(receive(a, x), \mu) \longrightarrow (\textbf{done}, \mu[x := m_1], \Phi[a = [m_2, \ldots, m_k]])}$$

$$global_s \quad \frac{\Delta(i) = (c, \mu) \quad (c, \mu, \Phi) \longrightarrow (c', \mu', \Phi')}{(\Delta, \Phi) \Longrightarrow (\Delta[i := (c', \mu')], \Phi')}$$

Fig. 3. Abstract Language Semantics

The Type System: Figure 4 shows the type system of the abstract language; its rules use an *identifier typing* Γ that maps identifiers to types. The typing rules enforce only the Denning restrictions [1]; in particular notice that we allow the guards of **while** loops to be H. Channels are restricted to carrying messages of one security classification from a specific process i to a specific process j and accordingly are typed $\Gamma(a) = \tau\ ch_{i,j}$ where τ is the security classification of the data that can travel in the channel, i is the source process and j is the destination process. So to enable full communications between processes i and j we need four channels with types $H\ ch_{i,j}$, $H\ ch_{j,i}$, $L\ ch_{i,j}$, and $L\ ch_{j,i}$. In a typing judgment

$$\Gamma, i \vdash c : \tau\ cmd$$

the process identifier i specifies which process command c belongs to; this is used to enforce the rule that only process i can send on a channel with type $\tau\ ch_{i,j}$ or receive on a channel with type $\tau\ ch_{j,i}$. We therefore say that a distributed system Δ is *well typed* if $\Delta(i) = (c, \mu)$ implies that $\Gamma, i \vdash c : \tau\ cmd$, for some τ.

Language Soundness: We now argue soundness properties for our language and type system, starting with some standard properties, whose proofs are straightforward.

Lemma 1 (Simple Security). *If $\Gamma, i \vdash e : \tau$, then e contains only variables of level τ or lower.*

$$\textit{sec} \quad \tau ::= H \mid L$$

$$\textit{phrase}\ \rho ::= \tau \mid \tau\ \textit{var} \mid \tau\ \textit{cmd} \mid \tau\ \textit{ch}_{i,j}$$

$$\textit{base} \quad L \subseteq H$$

$$\textit{cmd} \quad \frac{\tau \subseteq \tau'}{\tau'\ \textit{cmd} \subseteq \tau\ \textit{cmd}}$$

$$\textit{reflex} \quad \rho \subseteq \rho$$

$$\textit{trans} \quad \frac{\rho_1 \subseteq \rho_2 \quad \rho_2 \subseteq \rho_3}{\rho_1 \subseteq \rho_3}$$

$$\textit{subsump} \quad \frac{\Gamma, i \vdash p : \rho_1 \quad \rho_1 \subseteq \rho_2}{\Gamma, i \vdash p : \rho_2}$$

$$\textit{terminal}_t \quad \Gamma, i \vdash \textbf{done} : H\ \textit{cmd}$$

$$\textit{skip}_t \quad \Gamma, i \vdash \textbf{skip} : H\ \textit{cmd}$$

$$\textit{int}_t \quad \Gamma, i \vdash n : L$$

$$\textit{rval}_t \quad \frac{\Gamma(x) = \tau\ \textit{var}}{\Gamma, i \vdash x : \tau}$$

$$\textit{update}_t \quad \frac{\Gamma(x) = \tau\ \textit{var} \quad \Gamma, i \vdash e : \tau}{\Gamma, i \vdash x := e : \tau\ \textit{cmd}}$$

$$\textit{plus}_t \quad \frac{\Gamma, i \vdash e_1 : \tau \quad \Gamma, i \vdash e_2 : \tau}{\Gamma, i \vdash e_1 + e_2 : \tau}$$

$$\textit{if}_t \quad \frac{\Gamma, i \vdash e : \tau \quad \Gamma, i \vdash c_1 : \tau\ \textit{cmd} \quad \Gamma, i \vdash c_2 : \tau\ \textit{cmd}}{\Gamma, i \vdash \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 : \tau\ \textit{cmd}}$$

$$\textit{while}_t \quad \frac{\Gamma, i \vdash e : \tau \quad \Gamma, i \vdash c_1 : \tau\ \textit{cmd}}{\Gamma, i \vdash \textbf{while}\ e\ \textbf{do}\ c_1 : \tau\ \textit{cmd}}$$

$$\textit{compose}_t \quad \frac{\Gamma, i \vdash c_1 : \tau\ \textit{cmd} \quad \Gamma, i \vdash c_2 : \tau\ \textit{cmd}}{\Gamma, i \vdash c_1; c_2 : \tau\ \textit{cmd}}$$

$$\textit{send}_t \quad \frac{\Gamma(a) = \tau\ \textit{ch}_{i,j} \quad \Gamma, i \vdash e : \tau}{\Gamma, i \vdash \textit{send}(a, e) : \tau\ \textit{cmd}}$$

$$\textit{receive}_t \quad \frac{\Gamma(a) = \tau\ \textit{ch}_{j,i} \quad \Gamma(x) = \tau\ \textit{var}}{\Gamma, i \vdash \textit{receive}(a, x) : \tau\ \textit{cmd}}$$

Fig. 4. Abstract Language Type System

Lemma 2 (Confinement). *If $\Gamma, i \vdash c : \tau$ cmd, then c assigns only to variables of level τ or higher, and sends or receives only on channels of level τ or higher.*

Lemma 3 (Subject Reduction). *If $\Gamma, i \vdash c : \tau$ cmd and $(c, \mu, \Phi) \longrightarrow (c', \mu', \Phi')$, then $\Gamma, i \vdash c' : \tau$ cmd.*

We now turn to more interesting properties. We begin by defining *terminal* global configurations; these are simply configurations in which all processes have terminated:

Definition 1. *A global configuration (Δ, Φ) is* terminal *if for all i, $\Delta(i) = (\textbf{done}, \mu_i)$ for some μ_i.*

Notice that we do not require that Φ be an empty network memory—we allow it to contain unread messages.

Now we argue that, in spite of the nondeterminism of rule $global_s$, our distributed programs are *observationally deterministic* [7], in the sense that each program can reach at most one terminal configuration.

Theorem 1 (Observational Determinism). *Suppose that Δ is well typed and that $(\Delta, \Phi) \Longrightarrow^* (\Delta_1, \Phi_1)$ and $(\Delta, \Phi) \Longrightarrow^* (\Delta_2, \Phi_2)$, where (Δ_1, Φ_1) and (Δ_2, Φ_2) are terminal configurations. Then $(\Delta_1, \Phi_1) = (\Delta_2, \Phi_2)$.*

Proof. We begin by observing that the behavior of each process i is completely independent of the rest of the distributed system, with the sole exception of its *receive* commands. Thus if we specify the sequence of messages $[m_1, m_2, \ldots, m_n]$ that process i receives during its execution, then process i's behavior is completely determined. (Notice that the sequence $[m_1, m_2, \ldots, m_n]$ merges together all of the messages that process i receives on any of its input channels.)

We now argue by contradiction. Suppose that we can run from (Δ, Φ) to two different terminal configurations, (Δ_1, Φ_1) and (Δ_2, Φ_2). By the discussion above, it must be that some process receives a different sequence of messages in the two runs. So consider the *first* place in the second run $(\Delta, \Phi) \Longrightarrow^* (\Delta_2, \Phi_2)$ where a process i receives a different message than it does in the first run $(\Delta, \Phi) \Longrightarrow^* (\Delta_1, \Phi_1)$. But for this to happen, there must be another process j that earlier *sent* a different message to i than it does in the first run. (Note that this depends on the fact that, in a well-typed distributed system, any channel can be sent to by just one process and received from by just one process.) But for j to send a different message than in the first run, it must itself have received a different message earlier. This contradicts the fact that we chose the *first* place in the second run where a different message was received. □

We now wish to argue that well-typed distributed systems satisfy a termination-insensitive noninterference property. (We certainly need a termination-insensitive property since, under the Denning restrictions, H variables can affect termination.)

Definition 2. *Two memories μ and ν are* L-equivalent*, written $\mu \sim_L \nu$, if they agree on the values of all L variables. Similarly, two network memories Φ and Φ' are L-equivalent, also written $\Phi \sim_L \Phi'$, if they agree on the values of all L channels.*

Now we wish to argue that if we run a distributed system twice, using L-equivalent initial memories for each process, then, assuming that both runs terminate successfully, we must reach L-equivalent final memories for each process. A standard way to prove such a result is by establishing some sort of *low bisimulation* between the two runs. However this does not seem to be possible for our abstract language, because changing the values of H variables can affect when *receive* commands are able to be executed.

Figure 5 shows an example that illustrates the difficulty. Suppose we run this program twice, using two L-equivalent memories for Process 1, namely $[h_1 = 1, l_1 = 0]$ and $[h_1 = 0, l_1 = 0]$, and the same memory for Process 2, $[h_2 = 0, l_2 = 0]$. Under the first memory, Process 1 immediately sends on channel $a_{H,1,2}$, which then allows Process 2 to do its *receive* and then to assign to l_2 *before* Process 1 assigns to l_1. But under the second memory, Process 1 does not send on channel $a_{H,1,2}$ until after assigning to l_1, which means that the assignment to l_2 must come *after* the assignment to l_1. Thus

Process 1	**Process 2**
if h_1 **then** *send*$(a_{H,1,2}, 1)$ **else done**;	*receive*$(a_{H,1,2}, h_2)$;
$l_1 := 2$;	$l_2 := 3$
send$(a_{H,1,2}, 2)$	

Fig. 5. A difficult example for low bisimulation

the two runs are not low bisimilar. (Notice that the two runs are fine with respect to noninterference, however—in both cases we end up with $l_1 = 2$ and $l_2 = 3$.)

Because of this difficulty, we develop a different approach to noninterference, via the concepts of *stripping* and *fast simulation*, which were first used in [6]. Intuitively, the processes in Figure 5 contain H commands that are irrelevant to the L variables, except that they can cause *delays*. If we strip them out, we are left with

Process 1	**Process 2**
$l_1 := 2$	$l_2 := 3$

This shows what will happen to the L variables if the system terminates. We therefore introduce a *stripping* operation that eliminates all subcommands of type H *cmd*, so that the delays that such subcommands might have caused are eliminated. More precisely, we have the following definition:

Definition 3. *Let c be a well-typed command. We define $\lfloor c \rfloor =$ **done** if c has type H cmd; otherwise, define $\lfloor c \rfloor$ by*

- $\lfloor x := e \rfloor = x := e$
- $\lfloor \mathbf{if}\, e\, \mathbf{then}\, c_1\, \mathbf{else}\, c_2 \rfloor = \mathbf{if}\ e\ \mathbf{then} \lfloor\ c_1\ \rfloor \mathbf{else} \lfloor c_2 \rfloor$
- $\lfloor$**while** e **do** $c_1 \rfloor =$ **while** e **do** $\lfloor c_1 \rfloor$
- $\lfloor \mathit{send}(a, e) \rfloor = \mathit{send}(a, e)$
- $\lfloor \mathit{receive}(a, x) \rfloor = \mathit{receive}(a, x)$
- $\lfloor c_1; c_2 \rfloor = \begin{cases} \lfloor c_2 \rfloor & \text{if } c_1 : H \text{ cmd} \\ \lfloor c_1 \rfloor & \text{if } c_2 : H \text{ cmd} \\ \lfloor c_1 \rfloor; \lfloor c_2 \rfloor & \text{otherwise} \end{cases}$

Also, we define $\lfloor \mu \rfloor$ to be the result of deleting all H variables from μ, and $\lfloor \Phi \rfloor$ to be the result of deleting all H channels from Φ. We extend $\lfloor \cdot \rfloor$ to well-typed global configurations by $\lfloor (\Delta, \Phi) \rfloor = (\lfloor \Delta \rfloor, \lfloor \Phi \rfloor)$, where if $\Delta(i) = (c, \mu)$, then $\lfloor \Delta \rfloor (i) = (\lfloor c \rfloor, \lfloor \mu \rfloor)$.

We remark that stripping as defined in [6] replaces subcommands of type H *cmd* with **skip**; in contrast our new definition here aggressively *eliminates* such subcommands. Note also that $\lfloor \mu \rfloor \sim_L \mu$ and $\lfloor \Phi \rfloor \sim_L \Phi$. Now we have a simple lemma:

Lemma 4. *For any c, $\lfloor c \rfloor$ contains only L variables and channels.*

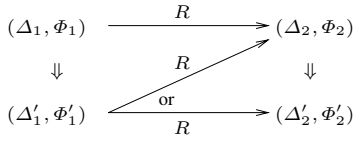
Proof. By induction on the structure of c. If c has type H *cmd*, then $\lfloor c \rfloor =$ **done**, which (vacuously) contains only L variables and channels.

If c does not have type H *cmd*, then consider the form of c. If c is $x := e$, then $\lfloor c \rfloor = x := e$. Since c does not have type H *cmd*, then by rule update_t we must have that x is a L variable and $e : L$, which implies by Simple Security that e contains only L variables. The cases of $\mathit{send}(a, e)$ and $\mathit{receive}(a, x)$ are similar. If c is **while** e **do** c_1, then $\lfloor c \rfloor =$ **while** e **do** $\lfloor c_1 \rfloor$. By rule while_t $e : L$, which implies by Simple Security that e contains only L variables and channels. And, by induction, $\lfloor c_1 \rfloor$ contains only L variables and channels. The cases of **if** e **then** c_1 **else** c_2 and $c_1; c_2$ are similar. □

Now the key result that we wish to establish is that $\lfloor (\Delta, \Phi) \rfloor$ can *simulate* (Δ, Φ), up to the final values of L variables. To this end we first adapt *fast simulation* from [6] (which in turn was based on strong and weak simulation in Baier et al [8]) to a nondeterministic (rather than probabilistic) setting.

Definition 4. *A binary relation R on global configurations is a* fast low simulation *with respect to $\Longrightarrow$ if whenever $(\Delta_1, \Phi_1) R (\Delta_2, \Phi_2)$ we have*

1. *(Δ_1, Φ_1) and (Δ_2, Φ_2) agree on the values of L variables and channels, and*
2. *if $(\Delta_1, \Phi_1) \Longrightarrow (\Delta_1', \Phi_1')$, then either $(\Delta_1', \Phi_1') R (\Delta_2, \Phi_2)$ or there exists (Δ_2', Φ_2') such that $(\Delta_2, \Phi_2) \Longrightarrow (\Delta_2', \Phi_2')$ and $(\Delta_1', \Phi_1') R (\Delta_2', \Phi_2')$. That is, (Δ_2, Φ_2) can match, in zero or one steps, any move from (Δ_1, Φ_1). In pictures:*



Viewing our stripping function $\lfloor \cdot \rfloor$ as a *relation*, we write $(\Delta_1, \Phi_1) \ \lfloor \cdot \rfloor \ (\Delta_2, \Phi_2)$ if $\lfloor (\Delta_1, \Phi_1) \rfloor = (\Delta_2, \Phi_2)$. Here is the key theorem about the stripping relation $\lfloor \cdot \rfloor$:

Theorem 2. *$\lfloor \cdot \rfloor$ is a* fast low simulation *with respect to $\Longrightarrow$.*

Proof. First, it is immediate from the definition of $\lfloor \cdot \rfloor$ that (Δ, Φ) and $\lfloor (\Delta, \Phi) \rfloor$ agree on the values of L variables and channels.

Next we must show that any move from (Δ, Φ) can be matched by $\lfloor (\Delta, \Phi) \rfloor$ in zero or one steps. Suppose that the move from (Δ, Φ) involves a step on process i. Then we must have $\Delta(i) = (c, \mu)$, $(c, \mu, \Phi) \longrightarrow (c', \mu', \Phi')$, and $\Delta' = \Delta[i := (c', \mu')]$. To show that $\lfloor (\Delta, \Phi) \rfloor$ can match this move in zero or one steps, note that $\lfloor (\Delta, \Phi) \rfloor = (\lfloor \Delta \rfloor, \lfloor \Phi \rfloor)$ and that $\lfloor \Delta \rfloor (i) = (\lfloor c \rfloor, \lfloor \mu \rfloor)$. Hence it suffices to show that either

$$(\lfloor c \rfloor, \lfloor \mu \rfloor, \lfloor \Phi \rfloor) = (\lfloor c' \rfloor, \lfloor \mu' \rfloor, \lfloor \Phi' \rfloor)$$

or else

$$(\lfloor c \rfloor, \lfloor \mu \rfloor, \lfloor \Phi \rfloor) \longrightarrow (\lfloor c' \rfloor, \lfloor \mu' \rfloor, \lfloor \Phi' \rfloor).$$

We argue this by induction on the structure of c.

If c has type H *cmd*, then $\lfloor c \rfloor =$ **done**. Also, by Confinement we have $\mu \sim_L \mu'$ and $\Phi \sim_L \Phi'$, which implies that $\lfloor \mu' \rfloor = \lfloor \mu \rfloor$, and $\lfloor \Phi' \rfloor = \lfloor \Phi \rfloor$. And by Subject Reduction we have $c' : H$ *cmd*, which implies that $\lfloor c' \rfloor =$ **done**. So the move $(c, \mu, \Phi) \longrightarrow (c', \mu', \Phi')$ is matched in zero steps by (**done**, $\lfloor \mu \rfloor$, $\lfloor \Phi \rfloor$).

If c does not have type H *cmd*, then consider the possible forms of c:

1. $c = x := e$. Here $\lfloor c \rfloor = c$. By *update*$_t$, $x : L$ *var* and $e : L$. So by Simple Security $\lfloor \mu \rfloor (e) = \mu(e)$, which implies that $\lfloor \mu \rfloor [x := \lfloor \mu \rfloor (e)] = \lfloor \mu [x := \mu(e)] \rfloor$. Hence the move $(c, \mu, \Phi) \longrightarrow$ (**done**, $\mu[x := \mu(e)], \Phi$) is matched by the move $(c, \lfloor \mu \rfloor, \lfloor \Phi \rfloor) \longrightarrow$ (**done**, $\lfloor \mu \rfloor [x := \lfloor \mu \rfloor (e)], \lfloor \Phi \rfloor$).
2. $c =$ *send*(a, e). Here $\lfloor c \rfloor = c$. By *send*$_t$, a is a low channel and $e : L$. Hence $\lfloor \Phi \rfloor (a) = \Phi(a) = [m_1, \ldots, m_k]$. Also, by Simple Security, $\lfloor \mu \rfloor (e) = \mu(e)$. Hence the move $(c, \mu, \Phi) \longrightarrow$ (**done**, $\mu, \Phi[a := [m_1, \ldots, m_k, \mu(e)]$) is matched by the move $(c, \lfloor \mu \rfloor, \lfloor \Phi \rfloor) \longrightarrow$ (**done**, $\lfloor \mu \rfloor, \lfloor \Phi \rfloor [a := [m_1, \ldots, m_k, \lfloor \mu \rfloor (e)]$).
3. $c =$ *receive*(a, x). Here $\lfloor c \rfloor = c$. By *receive*$_t$, a is a low channel and $x : L$ *var*. Hence $\lfloor \Phi \rfloor (a) = \Phi(a) = [m_1, \ldots, m_k]$, where $k \geq 1$. Therefore, the move $(c, \mu, \Phi) \longrightarrow$ (**done**, $\mu[x := m_1], \Phi[a := [m_2, \ldots, m_k]]$) is matched by the move $(c, \lfloor \mu \rfloor, \lfloor \Phi \rfloor) \longrightarrow$ (**done**, $\lfloor \mu \rfloor [x := m_1], \lfloor \Phi \rfloor [a := [m_2, \ldots, m_k]]$).

4. $c = \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2$. Here $\lfloor c \rfloor = \textbf{if } e \textbf{ then } \lfloor c_1 \rfloor \textbf{ else } \lfloor c_2 \rfloor$. By if_t, $e : L$ and by Simple Security $\mu(e) = \lfloor \mu \rfloor(e)$. So if $\mu(e) \neq 0$, then $(c, \mu, \Phi) \longrightarrow (c_1, \mu, \Phi)$ is matched by $(\lfloor c \rfloor, \lfloor \mu \rfloor, \lfloor \Phi \rfloor) \longrightarrow (\lfloor c_1 \rfloor, \lfloor \mu \rfloor, \lfloor \Phi \rfloor)$. The case when $\mu(e) = 0$ is similar.
5. $c = \textbf{while } e \textbf{ do } c_1$. Here $\lfloor c \rfloor = \textbf{while } e \textbf{ do } \lfloor c_1 \rfloor$. By $while_t$, $e : L$ and c_1 does not have type H *cmd*. By Simple Security, we have $\lfloor \mu \rfloor(e) = \mu(e)$. So in case $\mu(e) \neq 0$, then the move $(c, \mu, \Phi) \longrightarrow (c_1; \textbf{while } e \textbf{ do } c_1, \mu, \Phi)$ is matched by the move $(\lfloor c \rfloor, \lfloor \mu \rfloor, \lfloor \Phi \rfloor) \longrightarrow (\lfloor c_1 \rfloor; \textbf{while } e \textbf{ do } \lfloor c_1 \rfloor, \lfloor \mu \rfloor, \lfloor \Phi \rfloor)$. (This uses the fact that $\lfloor c_1; \textbf{while } e \textbf{ do } c_1 \rfloor = \lfloor c_1 \rfloor; \textbf{while } e \textbf{ do } \lfloor c_1 \rfloor$.) The case when $\mu(e) = 0$ is similar.
6. $c = c_1; c_2$. Here $\lfloor c \rfloor = \lfloor c_1; c_2 \rfloor$ has three possible forms: $\lfloor c_1 \rfloor; \lfloor c_2 \rfloor$, if neither c_1 nor c_2 has type H *cmd* (first subcase); $\lfloor c_2 \rfloor$, if $c_1 : H$ *cmd* (second subcase); or $\lfloor c_1 \rfloor$, if $c_2 : H$ *cmd* (third subcase).

   In the first subcase, neither c_1 nor c_2 has type H *cmd*. If the move from c is by the first rule $compose_s$, then $(c_1, \mu, \Phi) \longrightarrow (\textbf{done}, \mu', \Phi')$. By induction, this move can be matched by $(\lfloor c_1 \rfloor, \lfloor \mu \rfloor, \lfloor \Phi \rfloor)$ in zero or one steps. In fact it cannot be matched in zero steps—because c_1 does not have type H *cmd*, it is easy to see that $\lfloor c_1 \rfloor \neq \textbf{done}$. Hence we must have $(\lfloor c_1 \rfloor, \lfloor \mu \rfloor, \lfloor \Phi \rfloor) \longrightarrow (\textbf{done}, \lfloor \mu' \rfloor, \lfloor \Phi' \rfloor)$. It follows that $(c_1; c_2, \mu, \Phi) \longrightarrow (c_2, \mu', \Phi')$ is matched by $(\lfloor c_1 \rfloor; \lfloor c_2 \rfloor, \lfloor \mu \rfloor, \lfloor \Phi \rfloor) \longrightarrow (\lfloor c_2 \rfloor, \lfloor \mu' \rfloor, \lfloor \Phi' \rfloor)$. If instead the move from c is by the second rule $compose_s$, then $(c_1, \mu, \Phi) \longrightarrow (c_1', \mu', \Phi')$, where $c_1' \neq \textbf{done}$. By induction, $(\lfloor c_1 \rfloor, \lfloor \mu \rfloor, \lfloor \Phi \rfloor)$ can match this move, going in zero or one steps to $(\lfloor c_1' \rfloor, \lfloor \mu' \rfloor, \lfloor \Phi' \rfloor)$. Hence the move $(c_1; c_2, \mu, \Phi) \longrightarrow (c_1'; c_2, \mu', \Phi')$ can be matched by $(\lfloor c_1 \rfloor; \lfloor c_2 \rfloor, \lfloor \mu \rfloor, \lfloor \Phi \rfloor)$, going in zero or one steps to $(\lfloor c_1' \rfloor; \lfloor c_2 \rfloor, \lfloor \mu' \rfloor, \lfloor \Phi' \rfloor)$. A subtle point, however, is that even though c_1 does not have type H *cmd*, it is still possible that $c_1' : H$ *cmd*. In this case we *cannot* match by moving to $(\lfloor c_1' \rfloor; \lfloor c_2 \rfloor, \lfloor \mu' \rfloor, \lfloor \Phi' \rfloor)$, since here $\lfloor c_1'; c_2 \rfloor = \lfloor c_2 \rfloor \neq \lfloor c_1' \rfloor; \lfloor c_2 \rfloor = \textbf{done}; \lfloor c_2 \rfloor$. But here the match of the move from c_1 must actually be to $(\textbf{done}, \lfloor \mu' \rfloor, \lfloor \Phi' \rfloor)$, and it must be in one step (rather than zero) since $\lfloor c_1 \rfloor \neq \textbf{done}$. Hence in this case the move $(c_1; c_2, \mu, \Phi) \longrightarrow (c_1'; c_2, \mu', \Phi')$ is instead matched by the *first* rule $compose_s$: $(\lfloor c_1 \rfloor; \lfloor c_2 \rfloor, \lfloor \mu \rfloor, \lfloor \Phi \rfloor) \longrightarrow (\lfloor c_2 \rfloor, \lfloor \mu' \rfloor, \lfloor \Phi' \rfloor)$. [1]
   In the second subcase we have $c_1 : H$ *cmd* so $\lfloor c_1; c_2 \rfloor = \lfloor c_2 \rfloor$. If the move from c is by the first rule $compose_s$, then we must have $(c_1, \mu, \Phi) \longrightarrow (\textbf{done}, \mu', \Phi')$, where by Confinement $\mu' \sim_L \lfloor \mu \rfloor$ and $\Phi' \sim_L \lfloor \Phi \rfloor$. So the move $(c_1; c_2, \mu, \Phi) \longrightarrow (c_2, \mu', \Phi')$ is matched in zero steps by $(\lfloor c_2 \rfloor, \lfloor \mu \rfloor, \lfloor \Phi \rfloor)$. If instead the move from c is by the second rule $compose_s$, then we must have $(c_1, \mu, \Phi) \longrightarrow (c_1', \mu', \Phi')$, where by Confinement $\mu' \sim_L \lfloor \mu \rfloor$ and $\Phi' \sim_L \lfloor \Phi \rfloor$, and by Subject Reduction $c_1' : H$ *cmd*. Hence the move $(c_1; c_2, \mu, \Phi) \longrightarrow (c_1'; c_2, \mu', \Phi')$ is again matched in zero steps by $(\lfloor c_2 \rfloor, \lfloor \mu \rfloor, \lfloor \Phi \rfloor)$, since $\lfloor c_1'; c_2 \rfloor = \lfloor c_2 \rfloor$.
   Finally, the third subcase is similar to the first. □

Now we are ready to use these results in establishing our termination-insensitive noninterference result:

Theorem 3. *Let Δ_1 be a well-typed distributed program and let Δ_2 be formed by replacing each of the initial memories in Δ_1 with a L-equivalent memory. Let Φ_1 and*

[1] An example illustrating this situation when c is $(\textbf{if } 0 \textbf{ then } l := 1 \textbf{ else } h := 2);\ l := 3$. This goes in one step to $h := 2;\ l := 3$, which strips to $l := 3$. In this case, $\lfloor c \rfloor = (\textbf{if } 0 \textbf{ then } l := 1 \textbf{ else done});\ l := 3$, which goes *in one step* to $l := 3$.

Φ_2 be L-equivalent channel memories. Suppose that (Δ_1, Φ_1) and (Δ_2, Φ_2) can both execute successfully, reaching terminal configurations (Δ_1', Φ_1') and (Δ_2', Φ_2') respectively. Then the corresponding local memories of Δ_1' and Δ_2' are L-equivalent, and $\Phi_1' \sim_L \Phi_2'$.

Proof. By definition, $(\Delta_1, \Phi_1) \lfloor\cdot\rfloor \lfloor(\Delta_1, \Phi_1)\rfloor$ and $(\Delta_2, \Phi_2) \lfloor\cdot\rfloor \lfloor(\Delta_2, \Phi_2)\rfloor$. Hence, since $\lfloor\cdot\rfloor$ is a fast low simulation, we know that $\lfloor(\Delta_1, \Phi_1)\rfloor$ and $\lfloor(\Delta_2, \Phi_2)\rfloor$ can also execute successfully, and can reach terminal configurations whose local memories are L-equivalent to the corresponding memories of Δ_1' and Δ_2'. Moreover, by Theorem 1 we know that those terminal configurations are unique.

But $\lfloor(\Delta_1, \Phi_1)\rfloor$ is *identical* to $\lfloor(\Delta_2, \Phi_2)\rfloor$, since neither contains H variables or channels. Hence they must reach the *same* terminal configuration. It follows that the corresponding local memories of Δ_1' and Δ_2' are L-equivalent and that $\Phi_1' \sim_L \Phi_2'$. □

3 Towards a Concrete Implementation

In this section we explore the implementation of the abstract language in a concrete setting over a public network, including the abilities of the external adversary (Eve), the exploration of the appropriate network environment, and the characteristics of the security tools used to protect the data while it is being transmitted.

The abstract language's requirement of private channels limits its applicability to secure settings but we would like to implement our language in a more practical setting where communications between programs happen via a public network. We would like a setting like a wireless LAN but then we are faced with significant challenges to ensure confidentiality. Eavesdroppers can easily see all communications; what would it take to implement our language for distributed systems in a wireless environment? Clearly, secret data cannot be transmitted in a wireless LAN with any expectation of confidentiality as any computer with a receiver can get all the data that has been transmitted. Hence our *first requirement*, we need cryptography to hide the information being transmitted. Asymmetric cryptography seems to be the appropriate style for our setting.

Having decided on cryptography to hide the information that is being transmitted, we really want to encrypt only what is necessary to maintain the soundness of the distributed system, since encryption and decryption are expensive operations. In a wireless LAN, communications happen via electromagnetic signals which contain not only the message (payload) but also other information like source, destination, and data classification (header). Clearly we have to encrypt the payload but do we have to encrypt the header? In fact we do, for otherwise the language confidentiality would be lost as exemplified in Δ_4 of Figure 6. The channel used in both processes is a high channel (encrypted payload) yet an eavesdropper can still discern the value of the least bit of the secret h_1 by looking in the header of each packet for which process sends first; if Process 1 sends first then h_1 is odd and the least bit is 1. Therefore our *second requirement:* we have to encrypt the header and the payload of packets on high channels to prevent the leakage of secret information. Yet we are not done because surprisingly, this attack works even if the message sent is public and it is being sent on a public channel. Consider Δ_4 again and let's assume that we are encrypting all headers (secret and public)

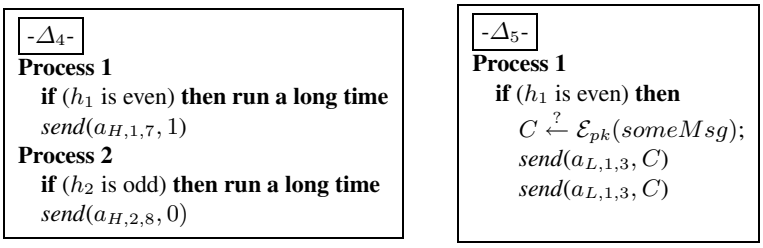


Fig. 6. Attacks (second wave)

and the secret payloads, but we are allowing the public data to be transmitted in the clear. This seems reasonable enough since the adversary will get all public data at the end of the execution. Nevertheless, if we observe a public value of 1 being transmitted first we will know with high probability that the least bit of h_1 is 1. Hence our *third requirement:* we have to encrypt all transmitted data.

We remark that if we had an active adversary which was able to drop packets, modify them and resend them, in addition to the attacks that we have seen she could modify packets to leak information and to affect the integrity of the distributed system. For example if the packet header was not encrypted she could change the packet classification from H to L thereby declassifying the payload which could cause the packet to be received by a low channel buffer in the receiving process. Then she can wait until the end of the execution to pick up the leaked secret from the process's public memory. This distinction may play a role in deciding what kind of security property will be necessary in our encryption scheme. Specifically a passive adversary might only require IND-CPA security while the active adversary will definitely require IND-CCA security. As an illustration of this distinction consider the following variation of the Warinschi attack on the Needham-Schroeder-(Lowe) protocol [9] where an IND-CPA scheme has the flaw where there is a function $C' := f(C)$ that takes an encrypted plaintext (like a packet header) and returns the ciphertext of an identical plaintext but with a certain location within it changed to L. This would not affect the security of the encryption scheme in any other way, i.e., an adversary would not be able to know anything about the content of the ciphertext but by simply substituting the header of any packet with C' and re-sending it, the adversary would be able to declassify the payload of the message.

But continuing with our analysis, are we done? We have decided to encrypt all data that is transmitted in the network, yet it is not enough to ensure confidentiality. Consider adversary Δ_5 of Figure 6, it encrypts a message and sends it two times if a secret is even. Meanwhile Eve scans every transmission waiting for two identical ciphertexts; if they are found she knows with high probability that the least bit of h_1 is 1, if all ciphers are distinct, it is 0. Therefore our *fourth requirement:* all transmitted data must be composed of freshly generated ciphertexts to ensure the confidentiality of our distributed systems.

Finally, there are two more attacks that we need to consider. The first one is the classical timing attack. If we know how long a typical execution step takes and can measure time intervals then we can leak information. Δ_6 of Figure 7 exemplifies this attack. If Eve is able to measure the time interval between the two transmissions she will have

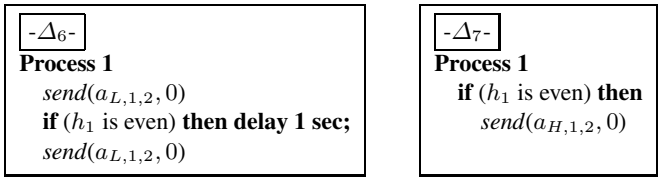


Fig. 7. Attacks (third wave)

the value of the least bit of h_1. A related timing attack is to count the number of message transmitted as in Δ_7 of Figure 7. In this attack, the last bit of h_1 is leaked by the transmission of one or zero packets. These attacks can be generalized to changing the statistics of packet transmission; for example, one can conceive an attack where the time distribution for packet transmission is uniform to leak a 0 and χ-square or normal for a 1. However, this area of study seems inappropriate for us to handle. A solution might be to impose a super-density of packet transmission at regular intervals, where only some of the packets are real messages. This would eliminate the problem but significantly increase the network bandwidth utilization. Another solution to this problem may the NRL-Pump [10,11] which is currently available at the local (government-owned) electronics shop. The pump obfuscates Eve's ability to measure the time between messages. It does this by inserting random delays based on an adaptive mechanism that adjusts the delays based on network traffic statistics. However, the pump cannot prevent timing channels based on the order of distinguishable messages.

To summarize, we not only have to somehow hide the meaning of messages, but also hide anything in a message that makes it distinguishable to Eve. Obviously, this includes payload, but also the message header, since the source or destination of a message can be used to distinguish it from another. Although L values are public, they cannot be seen within the network traffic. This is not a problem when computation happens within a processor (like in some multithreaded environments) because it is not reasonable that Eve would have access to the public memory in real time.

Soundness: Next we sketch a possible way to argue a computational noninterference property for our concrete language.

Random Transfer Language Definition and Soundness: First, we should be able to move our language, at the most basic level, from a nondeterministic to a probabilistic setting by constructing a subset language which we will call *Random Transfer Language*, and prove PNI on it. The PNI property on this language shall establish that if we allow only fresh-random traffic, the language is safe and sound.

Message Transfer Language Definition and Soundness: Then, we should be able to construct a *Message Transfer Language* and prove CNI on it. This language simply has the regular send command encrypt its payload before transmission on a public channel but keeps private channels for transmission of header information.

Header Transfer Language Definition and Soundness: Next, we should be able to construct *Header Transfer Language* and prove CNI on it. In this language the send command encrypt its header before transmission, but keeps private channels for transmission of the payload.

Hybrid Cryptographic Argument: Finally, we should be able to argue that since the Message and Header Transfer languages both satisfy CNI, the combination also should via a hybrid cryptographic argument.

4 Related Work

Peeter Laud [12,13] pioneers computationally secure information flow analysis with cryptography. Later, with Varmo Vene [14], they develop the first language and type system with cryptography and a computational security property. Recently, in [15], he proves a computational noninterference property on a type system derived from the work of Askarov et al [16].

In previous work on multithreaded languages [17,2,18,19,5], the type systems have strongly curtailed the language's expressive power in order to attain soundness. An expansion of these languages, with a rich set of cryptographic primitives, a treatment of integrity as well as confidentiality, and a subject close to ours is the work of Fournet and Rezk [20]. The primary differences between our papers are that their system does not handle concurrency and is subject to timing channels; on the other hand, their active adversary is more powerful having the ability to modify public data.

Another effort toward enhancing the usability of languages with security properties is the extensive functional imperative language $Aura$ [21]. The language maintains confidentiality and integrity properties of its constructs as specified by its label [22] by "packing" it using asymmetric encryption before declassification. The cryptographic layer is hidden to the programmer making it easier to use. This system uses static and runtime checking to enforce security. Using a different approach, Zheng and Myers [23] use a purely static type system to achieve confidentiality by splitting secrets under the assumption of non-collusion of repositories (e.g. key and data repositories). Under this model ciphertexts do not need to be public which allows relaxation of the type system while maintaining security. Further towards the practical end of the spectrum are efforts to provide assurance levels to software (as in EAL standard). In this line of work (Shaffer, Auguston, Irvine, Levin [24]) a security domain model is established and "real" programs are verified against it to detect flow violations.

Another paper close to ours is by Focardi and Centenaro [5]. It treats a multiprogrammed language and type system over asymmetric encryption and proves a noninterference property on it. The main differences are that their type system is more restrictive, requiring low guards on loops, and they use a formal methods approach rather than computational complexity.

5 Conclusion and Future Work

In this paper we have crafted an abstract language for distributed systems while maintaining a relaxed computational environment with private data, and we have argued that it has the noninterference property. We have explored the feasibility of implementing this language in a concrete setting where all communications happen via a public network with cryptography to protect confidentiality.

The obvious course for future work is to formalize these explorations and to prove computational noninterference on the concrete system. Another interesting area is to identify the environments where encryption schemes with weaker security (like IND-CPA) is sufficient to ensure soundness.

As the complexity of these languages increases, our reduction proofs may become unmanageable. One solution may be to use an automatic proving mechanism as in [25]. This work applies to security protocols in the computational model rather than languages. The tool works as a sequence of reductions towards a base that is easily proved to be secure, hence the original protocol is secure.

Acknowledgments

This work was partially supported by the National Science Foundation under grant CNS-0831114. We are grateful to Joshua Guttman, Pierpaolo Degano, and the FAST09 referees for helpful comments and suggestions.

References

1. Denning, D., Denning, P.: Certification of programs for secure information flow. Communications of the ACM 20(7), 504–513 (1977)
2. Smith, G., Volpano, D.: Secure information flow in a multi-threaded imperative language. In: Proceedings 25th Symposium on Principles of Programming Languages, San Diego, CA, January 1998, pp. 355–364 (1998)
3. Sabelfeld, A., Sands, D.: Probabilistic noninterference for multi-threaded programs. In: Proceedings 13th IEEE Computer Security Foundations Workshop, Cambridge, UK, July 2000, pp. 200–214 (2000)
4. Smith, G.: Improved typings for probabilistic noninterference in a multi-threaded language. Journal of Computer Security 14(6), 591–623 (2006)
5. Focardi, R., Centenaro, M.: Information flow security of multi-threaded distributed programs. In: PLAS 2008: Proceedings of the Third ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, pp. 113–124. ACM, New York (2008)
6. Smith, G., Alpízar, R.: Fast probabilistic simulation, nontermination, and secure information flow. In: Proc. 2007 ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, San Diego, California, June 2007, pp. 67–71 (2007)
7. Zdancewic, S., Myers, A.C.: Observational determinism for concurrent program security. In: Proceedings 16th IEEE Computer Security Foundations Workshop, Pacific Grove, California, June 2003, pp. 29–43 (2003)
8. Baier, C., Katoen, J.P., Hermanns, H., Wolf, V.: Comparative branching-time semantics for Markov chains. Information and Computation 200(2), 149–214 (2005)
9. Warinschi, B.: A computational analysis of the Needham-Schroeder-(Lowe) protocol. In: Proceedings 16th IEEE Computer Security Foundations Workshop, Pacific Grove, California, June 2003, pp. 248–262 (2003)
10. Kang, M.H., Moskowitz, I.S.: A pump for rapid, reliable, secure communication. In: CCS 1993: Proceedings of the 1st ACM Conference on Computer and Communications Security, pp. 119–129. ACM, New York (1993)
11. Kang, M.H., Moskowitz, I.S., Chincheck, S.: The pump: A decade of covert fun. In: 21st Annual Computer Security Applications Conference (ACSAC 2005), pp. 352–360. IEEE Computer Society, Los Alamitos (2005)

12. Laud, P.: Semantics and program analysis of computationally secure information flow. In: Sands, D. (ed.) ESOP 2001. LNCS, vol. 2028, pp. 77–91. Springer, Heidelberg (2001)
13. Laud, P.: Handling encryption in an analysis for secure information flow. In: Degano, P. (ed.) ESOP 2003. LNCS, vol. 2618, pp. 159–173. Springer, Heidelberg (2003)
14. Laud, P., Vene, V.: A type system for computationally secure information flow. In: Liśkiewicz, M., Reischuk, R. (eds.) FCT 2005. LNCS, vol. 3623, pp. 365–377. Springer, Heidelberg (2005)
15. Laud, P.: On the computational soundness of cryptographically masked flows. In: Proceedings 35th Symposium on Principles of Programming Languages, San Francisco, California (January 2008)
16. Askarov, A., Hedin, D., Sabelfeld, A.: Cryptographically-masked flows. In: Proceedings of the 13th International Static Analysis Symposium, Seoul, Korea, pp. 353–369 (2006)
17. Abadi, M., Fournet, C., Gonthier, G.: Secure implementation of channel abstractions. In: LICS 1998: Proceedings of the 13th Annual IEEE Symposium on Logic in Computer Science, Washington, DC, USA, p. 105. IEEE Computer Society, Los Alamitos (1998)
18. Smith, G.: Probabilistic noninterference through weak probabilistic bisimulation. In: Proceedings 16th IEEE Computer Security Foundations Workshop, Pacific Grove, California, June 2003, pp. 3–13 (2003)
19. Abadi, M., Corin, R., Fournet, C.: Computational secrecy by typing for the pi calculus. In: Kobayashi, N. (ed.) APLAS 2006. LNCS, vol. 4279, pp. 253–269. Springer, Heidelberg (2006)
20. Fournet, C., Rezk, T.: Cryptographically sound implementations for typed information-flow security. In: Proceedings 35th Symposium on Principles of Programming Languages, San Francisco, California (January 2008)
21. Jia, L., Vaughan, J.A., Mazurak, K., Zhao, J., Zarko, L., Schorr, J., Zdancewic, S.: Aura: a programming language for authorization and audit. In: Hook, J., Thiemann, P. (eds.) ICFP, pp. 27–38. ACM, New York (2008)
22. Vaughan, J., Zdancewic, S.: A cryptographic decentralized label model. In: IEEE Symposium on Security and Privacy, Oakland, California, pp. 192–206 (2007)
23. Zheng, L., Myers, A.C.: Securing nonintrusive web encryption through information flow. In: PLAS 2008: Proceedings of the third ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, pp. 125–134. ACM, New York (2008)
24. Shaffer, A.B., Auguston, M., Irvine, C.E., Levin, T.E.: A security domain model to assess software for exploitable covert channels. In: Erlingsson, Ú., Pistoia, M. (eds.) PLAS, pp. 45–56. ACM, New York (2008)
25. Blanchet, B.: A computationally sound mechanized prover for security protocols. In: SP 2006: Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P 2006), Washington, DC, USA, pp. 140–154. IEEE Computer Society Press, Los Alamitos (2006)