# Static Confidentiality Enforcement for Distributed Programs*

Andrei Sabelfeld[1] and Heiko Mantel[2]

[1] Department of Computer Science
Cornell University, Ithaca, NY 14853, USA, E-mail: andrei@cs.cornell.edu
[2] German Research Center for Artificial Intelligence (DFKI)
Stuhlsatzenhausweg 3, 66123 Saarbrücken, Germany, E-mail: mantel@dfki.de

**Abstract.** Preserving the confidentiality of data in a distributed system is an increasingly important problem of current security research. Distributed programming often involves message passing over a publicly observable medium, which opens up various opportunities for eavesdropping. Not only may the contents of messages sent on a *public* channel reveal confidential data, but merely observing the presence of a message on a channel for *encrypted* traffic may leak information. Another source of leaks is *blocking,* which may change the observable behavior of a process that attempts to receive on an empty channel.

In this article, we investigate the interplay between, on the one side, public, encrypted, and private (or hidden) channels of communication and, on the other side, blocking and nonblocking communication primitives for a simple multi-threaded language. We argue for timing-sensitive security and give a compositional timing-sensitive confidentiality specification. A key contribution of this article is a security-type system that statically enforces confidentiality. That the type system is not over-restrictive is exemplified by a typable distributed file-server program.

## 1 Introduction

Standard security infrastructure provides no satisfactory guarantees for preserving the *confidentiality* of data that is manipulated by computing systems. Consider a file-client program that accesses a distributed file server and manipulates confidential files via a publicly accessible network. In order to preserve the data's confidentiality, an array of security mechanisms is typically involved. First, an *access control* policy may enforce an authorized manipulation of files. Second, confidential data may be *encrypted* before it is sent on a public channel. Third, a *firewall* may protect the file server to restrict access from outside the subnetwork. While useful security building blocks, these standard techniques alone

* This research was supported by the Department of the Navy, Office of Naval Research, ONR Grant N00014-01-1-0968. Any opinions, findings, conclusions, or recommendations contained in this material are those of the authors and do not necessarily reflect the views of the Office of Naval Research. This work was supported, in part, by TFR, while the first author was at the Department of Computer Science, Chalmers University of Technology and University of Göteborg, Sweden. This work was supported, in part, by the German Research Foundation (DFG).

In *Proceedings of the 9th Static Analysis Symposium*, Madrid, Spain, September, 2002. LNCS 2477.

fail to guarantee that the confidentiality of data is preserved throughout the computation. The main reason is that these mechanisms ignore the *information flow* inside the program. Indeed, the file-client program that has legitimate access through the firewall, sends only encrypted data, and complies to access controls, may still leak information. For example, the program may contain the fragment if *fileScan*("foo") = *finApp* then *flag* := *true* where *flag* is a boolean variable (initialized to *false*) that signals whether to create a public temporary file containing a copy of the file "foo". If "foo" contains financial application data (*fileScan*("foo") = *finApp* holds), then this program successively surpasses the standard security mechanisms and introduces an undesired information flow. This is an example of an *implicit* [16] flow through a condition that depends on sensitive information (as opposed to an *explicit* flow performed by assigning sensitive data to a public variable). These leaks might be intended (when planted by a Trojan-horse program) or unintended (when caused by bugs, misconfiguration, debugging code, etc.). Syntactic scanning for malicious code (as realized by antivirus software) provides limited defense: rejecting a "black list" of syntactic patterns of known attacks does not preclude new attacks or semantic variations of the old ones.

An increasingly popular approach for identifying insecure information flow in programs is based on the notion of *noninterference* [20]. Suppose data, manipulated by a program, is partitioned into *high* (private) and *low* (public). According to noninterference, the program is secure iff high inputs do not interfere with low-observable behavior of the system (low outputs, timing, etc.). Originating from early work of Denning [15, 17] and Cohen [13, 14], a large body of work has followed the noninterference-based approach to confidentiality for various programming languages including [2, 22, 47, 49, 29, 6, 44, 45, 43, 12, 32, 46].[1] We follow this line of work in our definition of security[2] for a language enriched with *message passing*.

*Information flow in distributed languages.* As computing systems become increasingly connected, *multi-threaded* and *distributed* programming languages become increasingly important [8]. Typical distributed languages heavily rely on message passing (e.g., Erlang [9] and Java [21]). Client-server applications are examples of message-passing-based programs. For instance, a distributed file-server program may create a new thread for every incoming request. Such a request is a message (passed by the client program) to open, read/write, and close a file. The server responds with messages that grant or deny these accesses. Due to the distribution, messages often travel over a publicly observable medium. This opens up opportunities for compromising the confidentiality of data. Not only may messages' contents reveal confidential data, but merely observing the presence of a message may lead to an undesirable leak. The latter possibility requires particular care when message contents are protected by *encryption*. Another source of leaks is *blocking*, which may change the observable behavior of a

[1] For motivating noninterference for confidentiality we refer to [35, 50, 45].
[2] Security is restricted to confidentiality in the rest if the article.

process that attempts to receive on an empty channel. In a similar fashion, the *timing* behavior of the program may cause a flow that the attacker can observe. Hence, we design a noninterference-based security specification that, in addition to explicit and implicit flows, rejects these other flows.

*Static vs. dynamic confidentiality enforcement.* Note that the above code can be modified as $\mathit{flag} := \mathit{true}$; $\mathsf{if}\ \mathit{fileType}(\text{“foo”}) \neq \mathit{finApp}\ \mathsf{then}\ \mathit{flag} := \mathit{false}$ to perform the same leak. It is *non*execution of the assignment $\mathit{flag} := \mathit{false}$ that leads to this undesirable flow. As confidentiality, in general, is not a safety property [36, 48], the use of dynamic information-control mechanisms is rendered impractical as *all* potential execution paths must be monitored.

On the other hand, static analysis appears promising for enforcing security (e.g., [50, 22, 47, 6, 44, 28, 51, 11, 40]). Such an analysis is often formulated as a *security-type* system where security types correspond to the confidentiality level of data (such as high and low). A key contribution of this article is a security-type system that statically enforces confidentiality in a distributed system.

*Overview.* The rest of the paper is structured as follows. Section 2 explains main assumptions and features of the security model. Section 3 introduces a multi-threaded language with message passing. Section 4 presents a timing-sensitive security specification, scrutinizes the security of communication primitives wrt different types of channels, provides a series of examples, and concludes with compositionality results. Section 5 proposes a security-type system that enforces confidentiality. Section 6 gives an example of secure programming: a file server that relies on both multi-threading and message passing. Section 7 concludes.

## 2 Security Model

Distributed-language features have many implications for language security. The first step toward a tractable treatment is to clarify our assumptions behind the security model and motivate the model's intended features.

*Compositionality.* A prominent feature of a line of work [44, 45, 43] on noninterference for programs is the *compositionality* (or *hook-up* [34, 36, 31]) properties of security definitions. Such a property guarantees that when secure programs are plugged into an appropriate context then the resulting program must be secure. Hook-up properties provide an important foundation for modular system design. Thus, we aim to construct a compositional security property for a message-passing-enabled language.

*Timing-sensitive security.* Multi-threadedness (assuming a shared memory and execution on a single processor) has been a major focus of research in the context of noninterference-based confidentiality [22, 47, 49, 44, 43, 12, 32, 46]. Common to these studies is the observation that if a program's *timing* behavior depends on high data, then the *scheduler* may reflect this dependence on the values of low variables. Suppose $h$ and $l$ are high and low variables, i.e., variables that initially store high and low data, respectively. Consider the program $\mathsf{if}\ h = 1\ \mathsf{then}\ C_{long}\ \mathsf{else\ skip}$ where $C_{long}$ is a time-consuming computation. Clearly, the program's timing behavior may reveal information about $h$. This is an example of a *timing flow* [27]. Moreover, in a multi-threaded setting this kind of leak may be encoded into a program that leaks $h$ to $l$. Indeed, suppose $\|$ separates two threads in the two-threaded program $(\mathsf{if}\ h = 1\ \mathsf{then}\ C_{long}\ \mathsf{else\ skip}); l := 1 \ \ \| \ \ l := 0$. Under a typical fair scheduler (e.g., round-robin), if $h$ is 1 then it is likely that the assignment $l := 0$ will have been executed before the assignment $l := 1$; and, thus, the value of $h$ is likely to be copied to $l$. Note that the programs $l := 1$ and $l := 0$ are intuitively secure. Thus, accepting the original program as secure implies discarding the vital hook-up properties. This example illustrates that timing behavior, as observable by other threads, is an important ingredient for the security definition.

Our timing model follows previous work [44, 32, 43] in the assumption that the execution of each computation step takes a single unit of time. While this approach only roughly approximates the real timing behavior (which may depend on implementation- and hardware-specific factors as, e.g., caching [6]), it captures scheduler-based flows (as above) for a wide class of schedulers [44].

*Distribution.* In this article, we assume that threads are sequential programs; and multi-threading occurs at the level of local computation that operates on a shared memory. On the other hand, *processes* are potentially distributed such that each process has its own memory. Processes communicate by a communication network (among local computations) exchanging messages. Each process is potentially a multi-threaded program. Messages can be put onto a *channel* by sending commands and be taken out of a channel by receiving commands. Each channel is modeled by an *unordered list* (or a *finite multiset*) of messages, which adequately reflects the assumption that, in a distributed system, the order in which messages are delivered is not guaranteed.[3] For simplicity, we do not impose a particular discipline on using channels. However, depending on an application, it might be sensible to assume that, e.g., only one process can receive on a channel; or that a channel can be used only for point-to-point communication. Our security condition is independent of these assumptions.

*The adversary.* We assume that communication channels are partitioned into low, encrypted, and high channels. *Low* channels are observable by the attacker. Communication on low channels corresponds to, e.g., communication using standard Internet protocols such as TCP/IP and HTTP. Here, the traffic is vulnerable to eavesdropping. *Encrypted* channels are partly observable by the attacker. Namely, the attacker may observe the presence of a message, but not its contents. We adopt the (common) assumption that messages on such a channel are encrypted by an unbreakable encryption algorithm; and keys involved cannot

[3] An alternative assumption that a channel is a FIFO queue may be adequate for modeling communication channels at a high level of abstraction (e.g., pipelines in Unix). Our security specification can be easily adapted for this case (cf. [33]).

be compromised. (Ongoing research addresses cryptographic aspects of formal models of encryption [5, 28, 39].) Finally, *high* channels are secure connections between processes that are invisible to the attacker. Communication on high channels corresponds to, e.g., communication within a protected Intranet (an IP-based network of nodes behind a firewall). Here, the attacker cannot see the traffic.

The attacker has access to the low-observable part of the data and inter-process traffic. We assume that the machines in the network are trusted, although the code they run might come from an untrusted source. In this sense, the attacker is both *active*[4] (as a supplier of malicious code) and *passive* as an eavesdropper on the network. Our security-enforcement mechanism will reject potentially insecure programs from the active attacker so that running the code will not leak sensitive information to the passive attacker.

## 3 Multi-Threaded While-Language with Message Passing

This section presents the syntax and semantics of a simple multi-threaded language that conforms to the assumptions of the previous section.

*Local computation.* Syntax and semantics for local computations are adopted from [44]. The syntax of a *command* (or a *thread*) is given by the grammar in Figure 1. Let boolean expressions $B$ range over $BOOL$, arithmetic expressions $Exp$ range over $EXP$, variables $var$ range over $VAR$, values $val$ range over $VAL$, and commands $C, D$ range over $CMD$. Let $\vec{C}$ denote a vector of commands of the form $\langle C_1 \dots C_n \rangle$. Vectors $\vec{C}, \vec{D}$ range over $\vec{CMD} = \cup_{n \in \mathbb{N}} CMD^n$, the set of multi-threaded programs, and $cid$ ranges over $CID$, the set of channel identifiers.

A *configuration* $\langle\!| C, mem, \sigma |\!\rangle$ (or $\langle\!| \vec{C}, mem, \sigma |\!\rangle$) is a triple, consisting of a command $C \in CMD$ (or a vector of commands $\vec{C} \in \vec{CMD}$), a local memory $mem : VAR \to VAL$ and the third element $\sigma$ that accounts for interprocess communication (described below). The memory $mem$ is a finite mapping from variables to values. The set of variables is partitioned into high and low security classes. For simplicity (but without loss of generality), we will often assume that there is only one variable for each security class, $h$ and $l$, respectively. Further, we define *low-equality* on memories by: $mem_1 =_L mem_2$ iff the values of respective low variables for $mem_1$ and $mem_2$ are the same.

Each program executes under a shared memory on a single processor (or in a single process) such that at most one thread is active at any given point of time. The small-step semantics is given by transitions between configurations. The deterministic part of the semantics is defined by the transition rules in Figure 2, omitting natural rules for skip, assignment, if and while for lack of space (cf. [44]). Arithmetic and boolean expressions are executed atomically by $\downarrow$ transitions. The local $\rightharpoonup$-transitions are deterministic. The general form of a deterministic transition is either $\langle\!| C, mem, \sigma |\!\rangle \rightharpoonup \langle\!| \langle\rangle, mem', \sigma' |\!\rangle$, which means

[4] We do not consider *integrity* properties and thus active attackers faking messages are not relevant.

$$
\begin{aligned}
\mathit{CMD} ::= {} & \mathsf{skip} \mid \mathit{VAR} := \mathit{EXP} \mid \mathit{CMD}_1; \mathit{CMD}_2 \mid \mathsf{if}\ \mathit{BOOL}\ \mathsf{then}\ \mathit{CMD}_1\ \mathsf{else}\ \mathit{CMD}_2\\
& \mid \mathsf{while}\ \mathit{BOOL}\ \mathsf{do}\ \mathit{CMD} \mid \mathsf{fork}(\mathit{CMD}, \vec{\mathit{CMD}}) \mid \mathsf{send}(\mathit{CID}, \mathit{EXP})\\
& \mid \mathsf{receive}(\mathit{CID}, \mathit{VAR}) \mid \mathsf{if\text{-}receive}(\mathit{CID}, \mathit{VAR}, \mathit{CMD}_1, \mathit{CMD}_2)
\end{aligned}
$$

**Fig. 1.** Command syntax

$$[\mathrm{Seq}_1]\quad \frac{\langle\!| C_1, mem, \sigma |\!\rangle \twoheadrightarrow \langle\!| \langle\rangle, mem', \sigma |\!\rangle}{\langle\!| C_1; C_2, mem, \sigma |\!\rangle \twoheadrightarrow \langle\!| C_2, mem', \sigma |\!\rangle}$$

$$[\mathrm{Seq}_2]\quad \frac{\langle\!| C_1, mem, \sigma |\!\rangle \twoheadrightarrow \langle\!| C_1'\vec{D}, mem', \sigma |\!\rangle}{\langle\!| C_1; C_2, mem, \sigma |\!\rangle \twoheadrightarrow \langle\!| (C_1'; C_2)\vec{D}, mem', \sigma |\!\rangle}$$

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

$$[\mathrm{Send}]\quad \frac{Exp \downarrow^{mem} val \qquad vals = \sigma(cid)}{\langle\!| \mathsf{send}(cid, Exp), mem, \sigma |\!\rangle \twoheadrightarrow \langle\!| \langle\rangle, mem, \sigma[cid \mapsto val.vals] |\!\rangle}$$

$$[\mathrm{Rcv}]\quad \frac{\sigma(cid) = vals.val.vals'}{\langle\!| \mathsf{receive}(cid, var), mem, \sigma |\!\rangle \twoheadrightarrow \langle\!| \langle\rangle, mem[var \mapsto val], \sigma[cid \mapsto vals.vals'] |\!\rangle}$$

$$[\mathrm{IfRF}]\quad \frac{\sigma(cid) = \langle\rangle}{\langle\!| \mathsf{if\text{-}receive}(cid, var, C_1, C_2), mem, \sigma |\!\rangle \twoheadrightarrow \langle\!| C_2, mem, \sigma |\!\rangle}$$

$$[\mathrm{IfRT}]\quad \frac{\sigma(cid) = vals.val.vals'}{\langle\!| \mathsf{if\text{-}receive}(cid, var, C_1, C_2), mem, \sigma |\!\rangle \twoheadrightarrow \langle\!| C_1, mem[var \mapsto val], \sigma[cid \mapsto vals.vals'] |\!\rangle}$$

**Fig. 2.** Small-step deterministic semantics of commands

$$[\mathrm{Pick}]\quad \frac{\langle\!| C_i, mem, \sigma |\!\rangle \twoheadrightarrow \langle\!| \vec{C}, mem', \sigma' |\!\rangle}{\langle\!| \langle C_1 \dots C_n \rangle, mem, \sigma |\!\rangle \rightarrow \langle\!| \langle C_1 \dots C_{i-1}\vec{C}C_{i+1} \dots C_n \rangle, mem', \sigma' |\!\rangle}$$

$$[\mathrm{Step}]\quad \frac{\langle\!| \vec{C}_k, mem_k, \sigma |\!\rangle \rightarrow \langle\!| \vec{C}_k', mem_k', \sigma' |\!\rangle}{\lhd (\vec{C}_1, mem_1), \dots, (\vec{C}_n, mem_n); \sigma \rhd \twoheadrightarrow \lhd (\vec{C}_1, mem_1), \dots, (\vec{C}_k', mem_k'), \dots, (\vec{C}_n, mem_n); \sigma' \rhd}$$

**Fig. 3.** Concurrent semantics of programs

$$
\begin{aligned}
\mathsf{synch\text{-}send}(cid, Exp) &= \mathsf{receive}(destReady[cid], readyVar); \mathsf{send}(cid, Exp);\\
&\phantom{=} \mathsf{receive}(destAck[cid], ackVar)\\
\mathsf{synch\text{-}receive}(cid, var) &= \mathsf{send}(destReady[cid], \text{“ready”}); \mathsf{receive}(cid, var);\\
&\phantom{=} \mathsf{send}(destAck[cid], \text{“got it”})
\end{aligned}
$$

**Fig. 4.** Synchronous communication primitives

termination with the final memory $mem'$, or $\langle\!| C, mem, \sigma |\!\rangle \twoheadrightarrow \langle\!| C'\vec{D}, mem', \sigma' |\!\rangle$. Here, one step of computation starting with command $C$ in a memory $mem$ gives a new main thread $C'$, a vector $\vec{D}$ of spawned threads, a new memory $mem'$ and new channel status $\sigma'$. The command $\mathsf{fork}(C, \vec{D})$ dynamically creates a new vector $\vec{D}$ of threads that run in parallel with the main thread $C$. This has the effect of adding the vector $\vec{D}$ to the configuration.

The rule [Pick] in Figure 3 defines the concurrent semantics within a program. Whenever the scheduler picks a thread $C_i$ for execution, a $\rightarrow$-transition takes place updating the command pool and the rest of the configuration according to a (small) computation step of $C_i$.

*Communication primitives and global computation.* We now present the communication primitives and global semantics, which we adapt from [33]. Let channel contents range over *CHVAL*, the set of unordered lists over *VAL*. Given a channel id *cid*, the *channel status function* $\sigma : CID \rightarrow CHVAL$ returns the unordered list of messages that are currently waiting on *cid*.

The command $\mathsf{send}(cid, Exp)$ sends the value of an expression *Exp* on a channel *cid*. We distinguish two receiving primitives. *Blocking* $\mathsf{receive}(cid, var)$ blocks until it receives a value on the channel *cid*. Once the value is received, the variable *var* is set to that value. *Nonblocking* receive $\mathsf{if\text{-}receive}(cid, var, C_1, C_2)$ always continues execution. If the channel *cid* is nonempty then *var* is set to the received value and execution continues with the command $C_1$. Otherwise execution continues with the command $C_2$. Note that the order of delivery does not necessarily coincide with the order in which messages were sent. This accurately models our assumption (cf. Section 2) implied by the distributed nature of the system. Note that the primitive send is *asynchronous* in the sense that it does not wait for the message to be received. A *synchronous* version synch-send blocks until the message has been received by a synch-receive on the same channel. Synchronous communication (synch-send and synch-receive) can be achieved by using asynchronous communication primitives. Figure 4 displays how this can be done in the simple case of at most one sender and at most one receiver for each channel $cid \in CID$ (cf. [8]). It uses auxiliary arrays of channels *destReady* and *destAck*. Care must be taken when constructing such an implementation when multiple readers/writers are possible. (For example, the encoding of Figure 4 is then not suitable as a *destAck* message may be received by a sender whose message is still in transit). For our purposes, the key observation is that both synch-send and synch-receive use receive in their implementations.

Given a (possibly distributed) collection of programs $\vec{C_1}, \ldots, \vec{C_n}$, in which each program executes on its own memory $mem_1, \ldots, mem_n$, respectively, a *global* configuration $\lhd(\vec{C_1}, mem_1), \ldots, (\vec{C_n}, mem_n); \sigma\rhd$ consists of two components. The first component is a sequence of pairs each containing a program and its memory. The second component is the channel status function $\sigma$. Nondeterministic $\twoheadrightarrow$-transitions on global configurations are defined by the rule [Step] in Figure 3. The rule [Step] is similar to the rule [Pick]. It ensures that a global transition takes place whenever a local transition occurs.

## 4 Security Specification

Recall that noninterference means that low-observable behavior is unchanged as high inputs are varied. The indistinguishability of behavior for the attacker can be represented naturally by the notion of *bisimulation* (e.g, [18, 44, 46, 12]). Following our justification of timing-sensitive security in the context of concurrent and distributed programming in Section 2, we adopt the timing-sensitive *strong low-bisimulation* [44] that can be defined on multi-threaded programs with [43] and without synchronization [44]. If two commands might have a different timing behavior depending on high data then they are not low-bisimilar. This definition has been extended for a language with message passing, but only with high and low channels for communication [33].

### 4.1 Timing-sensitive security definition

In order to lift low-bisimulation to handle encrypted channels we extend low-equality $=_L$ (defined earlier on memories) to relate channel status functions that agree on their low-observable arguments. Let $dom : CID \to \{high, enc, low\}$ be a function that given a channel id $cid$ returns its security level. Formally, define for $\sigma_1, \sigma_2 : CID \to CHVAL$:

$$\begin{aligned}\sigma_1 =_L \sigma_2 \iff (\forall cid \in CID.\ & dom(cid) = low \Longrightarrow \sigma_1(cid) = \sigma_2(cid) \wedge \\ & dom(cid) = enc \Longrightarrow |\sigma_1(cid)| = |\sigma_2(cid)|)\end{aligned}$$

where $|\sigma(cid)|$ is the number of messages on channel $cid$. This suits our assumption that for the attacker: (i) low channels are fully observable, (ii) only the size of encrypted channels is observable, and (iii) *high* channels are not observable.

**Definition 1** *Define* strong low-bisimulation $\approxeq_L$ *to be the union of all symmetric relations $R$ on command pools (programs) of equal size for which whenever $\langle C_1 \dots C_n\rangle\ R\ \langle D_1 \dots D_n\rangle$ then for all $mem_1, mem_2, \sigma_1, \sigma_2, i$ we have*

$$\langle\!| C_i, mem_1, \sigma_1 |\!\rangle \twoheadrightarrow \langle\!| \vec{C'}, mem'_1, \sigma'_1 |\!\rangle \wedge mem_1 =_L mem_2 \wedge \sigma_1 =_L \sigma_2 \Longrightarrow \exists \vec{D'}, mem'_2, \sigma'_2.$$
$$\langle\!| D_i, mem_2, \sigma_2 |\!\rangle \twoheadrightarrow \langle\!| \vec{D'}, mem'_2, \sigma'_2 |\!\rangle \wedge mem'_1 =_L mem'_2 \wedge \sigma'_1 =_L \sigma'_2 \wedge \vec{C'}\ R\ \vec{D'}$$

Intuitively, two programs $\langle C_1 \dots C_n\rangle$ and $\langle D_1 \dots D_n\rangle$ are low-bisimilar iff (i) they are command vectors of the same size and (ii) for each pair of respective commands $C_i$ and $D_i$ occurring in the same position $i$, varying the high parts of memories, the contents of encrypted channels (but not size) and the contents of high channels (including the size) at any point of a computation does not introduce any difference between the low parts of the memories, the sizes of encrypted channels and the contents of low channels throughout the computation. Despite any high variation such two commands will still execute in lock step, i.e., their timing behavior is the same. We are ready to state the security specification.

**Definition 2** *A program $\vec{C}$ is* secure *if and only if $\vec{C} \approxeq_L \vec{C}$.*

While one can prove that the relation $\approxeq_L$ is transitive, it is not reflexive. For example, the insecure program $l := h$ is not $\approxeq_L$-related to itself, as the low-equality of memories can be broken by a computation step.

### 4.2 Security of communication primitives

This section is devoted to the communication primitives and their effect on security. Of particular interest is the identification of *secure contexts* for hook-up properties (cf. Section 2) involving communication primitives. Inserting secure programs in such a context should result in a secure program. In the next section, we will derive a security-type system from the hook-up properties. Define an expression $Exp$ to be *low* iff $\forall mem_1, mem_2.\ mem_1 =_L mem_2 \Longrightarrow \exists n.\ Exp \downarrow^{mem_1} n \wedge Exp \downarrow^{mem_2} n$. Otherwise, the expression is *high*. In the rest of this section, $Exp_h$ ranges over high expressions while $Exp$ ranges over arbitrary expressions.

*Send.* Sending on either a high or an encrypted channel is a secure command, because send is nonblocking. Indeed, executing $\mathsf{send}(cid, Exp)$ after varying the high part of the memory and channel status function will always result in low-equal memories and low-equal channel status functions. This is true for both high and encrypted channels: low-level observations remain unchanged in the case of sending on a high channel. In the case of an encrypted channel, the size of the channel will always be increased by 1 independently of the variation of high data. To show that such a program is secure we, according to Definition 1, need to construct a symmetric relation that makes this program low-bisimilar to itself. One suitable relation is the relation $\{(\mathsf{send}(cid, Exp), \mathsf{send}(cid, Exp)), (\langle\rangle, \langle\rangle)\}$. This proves that $\mathsf{send}(cid, Exp)$ is a secure ground context for any expression $Exp$. This and following examples of communication primitives and their security are collected in Figure 7 in Appendix A.

However, if a send occurs in a branch of a *high conditional* (i.e., a conditional whose guard is a high expression), then sending on an encrypted channel in one branch and not sending in the other (e.g., $\mathsf{if}\ Exp_h\ \mathsf{then}\ \mathsf{send}(cid, Exp)\ \mathsf{else}\ \mathsf{skip}$) makes low-observable behavior different. Namely, the size of the channel will be different when varying the initial value of $h$. Formally, take $mem_1$ and $mem_2$ such that $mem_1 =_L mem_2$, $Exp_h \downarrow^{mem_1} \mathit{false}$, $Exp_h \downarrow^{mem_2} \mathit{true}$ (which is possible because $Exp_h$ is high), and $\sigma = \lambda(cid).\langle\rangle$. Since $\langle\!|\mathsf{if}\ Exp_h\ \mathsf{then}\ \mathsf{send}(cid, Exp)\ \mathsf{else}\ \mathsf{skip}, mem_1, \sigma|\!\rangle \rightarrow \langle\!|\mathsf{skip}, mem_1, \sigma|\!\rangle$ and $\langle\!|\mathsf{if}\ Exp_h\ \mathsf{then}\ \mathsf{send}(cid, Exp)\ \mathsf{else}\ \mathsf{skip}, mem_2, \sigma|\!\rangle \rightarrow \langle\!|\mathsf{send}(cid, Exp), mem_2, \sigma|\!\rangle$ holds, for the initial program to be secure it must be the case that $\mathsf{skip} \approxeq_L \mathsf{send}(cid, Exp)$. However, we have $\langle\!|\mathsf{skip}, mem_1, \sigma|\!\rangle \rightarrow \langle\!|\langle\rangle, mem_1, \sigma|\!\rangle$ and $\langle\!|\mathsf{send}(cid, Exp), mem_2, \sigma|\!\rangle \rightarrow \langle\!|\langle\rangle, mem_2, \sigma[cid \mapsto \langle n\rangle]|\!\rangle$, where $Exp \downarrow^{mem_2} n$. Clearly, $\sigma \neq_L \sigma[cid \mapsto \langle n\rangle]$ for any $n$ because $dom(cid) = enc$. Note that the same program for a high channel is perfectly secure.

A similar argument explains why sending on different channels depending on a high condition (e.g., $\mathsf{if}\ Exp_h\ \mathsf{then}\ \mathsf{send}(cid, Exp)\ \mathsf{else}\ \mathsf{send}(cid', Exp'))$ is insecure. Only when sending (possibly different) values on the same channel in both branches, may a command be secure (e.g., $\mathsf{if}\ Exp_h\ \mathsf{then}\ \mathsf{send}(cid,Exp)\ \mathsf{else}\ \mathsf{send}(cid, Exp'))$ for an encrypted channel. Sending $Exp_h$ on a low channel $cl$ is, naturally, insecure (e.g., $\mathsf{send}(cl, Exp_h)$).

*Receive.* Receiving on a high channel (e.g., $\mathsf{receive}(cid, h)$) is insecure as it may introduce blocking if the channel is empty. On the other hand, receiving on an encrypted channel is secure because the number of messages on an encrypted channel is the same for two low-equal channel status functions. As is the case with conditional sending, conditional receiving is insecure for encrypted channels (e.g., if $Exp_h$ then receive($cid, h$) else skip and also if $Exp_h$ then receive($cid, h$) else receive($cid', h$)) unless the same channel is used (e.g., if $Exp_h$ then receive($cid, h$) else receive($cid, h'$)). Using a low variable for storing the data received on a high or encrypted channel (e.g., receive($cid, l$)) is, of course, insecure.

Nonblocking receive if-receive($cid, h, C_1, C_2$) on an encrypted channel is secure provided both branches $C_1$ and $C_2$ are secure programs. Indeed, what branch was taken is observable by the attacker because it is observable whether the channel *cid* is empty. In case *cid* is high, the stronger condition $C_1 \approxeq_L C_2$ (which implies that $C_1$ and $C_2$ are secure by the transitivity of the relation $\approxeq_L$) is imposed to ensure the security of the overall program. Such a condition guarantees that the behavior of the branches is indistinguishable for the attacker.

*Synchronous communication.* Both synch-send and synch-receive inherit their security properties from the receive that is used in their implementations. This means that neither synchronous sending nor receiving is secure on a high channel. For an encrypted channel *cid*, the programs synch-send($cid, h$), synch-receive($cid, h$), if $Exp_h$ then synch-send($cid, Exp$) else synch-send($cid, Exp'$), and if $Exp_h$ then synch-receive($cid, h$) else synch-receive($cid, h'$) are secure.

### 4.3 Lessons learned

Essential implications for the programmer from our consideration of the security of communication primitives wrt different types of channels can be briefly summarized as follows:

- High channels allow for liberal use of send in the branches of high conditionals. However, receive on a high channel is not secure due to the potential introduction of blocking. As a result, the programmer is restricted to the nonblocking if-receive which masks the presence of messages on the channel.
- Encrypted channels are appropriate for both send and receive, but restrictive on high conditionals. Thus, when the occurrence of a communication of some high data does not depend on any high data, it is an appropriate channel for an efficient implementation. Indeed, as receive is a secure primitive, there is no need for the busy-waiting loops with the if-receive primitive.
- if-receive is a secure primitive for both high and encrypted channels, but for encrypted channels receive is preferable over the inefficient busy-waiting loops with if-receive.
- Each of the synchronous primitives synch-send and synch-receive inherit their restrictiveness from receive. This implies that neither synchronous sending nor synchronous receiving is a secure command for a high channel.
- Confidentiality might be compromised if communication primitives operating on encrypted or low channels are used in the branches of a high if or if-receive that receives on a high channel. This is similar to implicit flows that are the effect of an assignment to $l$ in the branches of a high if.

### 4.4 Hook-up properties

Supported by the intuition of Sections 4.2 and 4.3, we present the compositionality result for the complete language. This result extends the secure congruence theorem for a multi-threaded language [44] to contexts that preserve security in the presence of message passing. The proof of the hook-up result is conducted by the "up to" technique [37]. Let $[\bullet]$ be a hole for a command. A context $\mathbb{C}[\bullet_1, \ldots, \bullet_n]$ for some $n \geq 2$ is *secure* iff it has one of these forms:

$$
\begin{aligned}
\mathbb{C}[\bullet_1, \ldots, \bullet_n] ::= \ & \mathsf{skip} \mid h := Exp \mid l := Exp \ (Exp \text{ is low}) \mid [\bullet_1]; [\bullet_2] \\
& \mid \mathsf{if}\ B\ \mathsf{then}\ [\bullet_1]\ \mathsf{else}\ [\bullet_2]\ (B \text{ is low}) \mid \mathsf{while}\ B\ \mathsf{do}\ [\bullet_1]\ (B \text{ is low}) \\
& \mid \mathsf{fork}([\bullet_1], \langle [\bullet_2] \ldots [\bullet_n] \rangle) \mid \mathsf{send}(cid, Exp)\ (dom(cid) = high) \\
& \mid \mathsf{send}(cid, Exp),\ \mathsf{receive}(cid, h)\ (dom(cid) = enc) \\
& \mid \mathsf{send}(cid, Exp),\ \mathsf{receive}(cid, var)\ (dom(cid) = low,\ Exp \text{ is low}) \\
& \mid \mathsf{if\text{-}receive}(cid, h, [\bullet_1], [\bullet_2])\ (dom(cid) = enc) \\
& \mid \mathsf{if\text{-}receive}(cid, var, [\bullet_1], [\bullet_2])\ (dom(cid) = low)
\end{aligned}
$$

**Theorem 1 (Hook-up)** *If for some $n \geq 2$ the context $\mathbb{C}[\bullet_1, \ldots, \bullet_n]$ is a secure context and $C_1, \ldots, C_n$ are secure then $\mathbb{C}[C_1, \ldots, C_n]$ is secure. If $\mathbb{C}[\bullet_1, \bullet_2] =$ $\mathsf{if}\ B\ \mathsf{then}\ [\bullet_1]\ \mathsf{else}\ [\bullet_2]$ ($B$ is high) or $\mathbb{C}[\bullet_1, \bullet_2] = \mathsf{if\text{-}receive}(cid, h, [\bullet_1], [\bullet_2])$ (such that $dom(cid) = high$), then $\mathbb{C}[C_1, C_2]$ is secure provided $C_1 \approxeq_L C_2$.*

## 5 Type-based Security Analysis

This section presents an automatic compositional analysis for enforcing program confidentiality, extending previous approaches [6, 44] to handle communication primitives. Due to the compositional nature of the type system, both designing it and proving it to be sound are greatly facilitated by the hook-up results.

*The type system.* The analysis is based on a type system that transforms a given program into a new program. Either the initial program is rejected (due to explicit, implicit or other insecure information leaks) or it might be accepted by the system and transformed into a program that is also free of timing leaks. The transformation rules have the form $\vec{C} \hookrightarrow \vec{C'} : \vec{Sl}$, where $\vec{C}$ is a program, $\vec{C'}$ is the result of its transformation and $\vec{Sl}$ is the type of $\vec{C'}$. The type $\vec{Sl}$ is $\vec{C'}$'s *low slice*. The low slice $\vec{Sl}$ has no occurrences of $h$ and models the timing behavior of $\vec{C'}$, as observable by other threads/processes.

Any expression can by typed *high* (as it might depend on $h$). On the other hand, only expressions that have no occurrences of $h$ are typed *low*. (These expressions can be safely used in assignments to $l$.) Typing rules for expressions and communication primitives are presented in Figure 5. Typing rules for other language constructs are like in [44]. The rule $[\mathrm{S}_{high}]$ types a send on a high channel with the low slice skip. Although receive on a high channel is not typable, if-receive can be typed after cross-copying the branches (as for if on a high conditional

$$[\mathrm{Exp}] \quad Exp : high \qquad \frac{h \notin Vars(Exp)}{Exp : low}$$

————————————————————————————————— ***cid* is *high***

$$[\mathrm{S}_{high}] \quad \mathsf{send}(cid, Exp) \hookrightarrow \mathsf{send}(cid, Exp) : \mathsf{skip}$$

$$[\mathrm{IR}_{high}] \quad \frac{C_1 \hookrightarrow C_1' : Sl_1 \quad C_2 \hookrightarrow C_2' : Sl_2 \quad al(Sl_1) = al(Sl_2) = false}{\mathsf{if\text{-}receive}(cid,h,C_1,C_2) \hookrightarrow \mathsf{if\text{-}receive}(cid,h,C_1';Sl_2,Sl_1;C_2') : \mathsf{skip};\, Sl_1;\, Sl_2}$$

————————————————————————————————— ***cid* is *enc***

$$[\mathrm{S}_{enc}] \quad \mathsf{send}(cid, Exp) \hookrightarrow \mathsf{send}(cid, Exp) : \mathsf{send}(cid, 0)$$

$$[\mathrm{R}_{enc}] \quad \mathsf{receive}(cid, h) \hookrightarrow \mathsf{receive}(cid, h) : \mathsf{receive}(cid, \hat{h})$$

$$[\mathrm{IR}_{enc}] \quad \frac{C_1 \hookrightarrow C_1' : Sl_1 \quad C_2 \hookrightarrow C_2' : Sl_2}{\mathsf{if\text{-}receive}(cid,h,C_1,C_2) \hookrightarrow \mathsf{if\text{-}receive}(cid,h,C_1',C_2') : \mathsf{if\text{-}receive}(cid,\hat{h},Sl_1,Sl_2)}$$

$$[\mathrm{SS}_{enc}] \quad \mathsf{synch\text{-}send}(cid, Exp) \hookrightarrow \mathsf{synch\text{-}send}(cid, Exp) : \mathsf{synch\text{-}send}(cid, 0)$$

$$[\mathrm{SR}_{enc}] \quad \mathsf{synch\text{-}receive}(cid, h) \hookrightarrow \mathsf{synch\text{-}receive}(cid, h) : \mathsf{synch\text{-}receive}(cid, \hat{h})$$

————————————————————————————————— ***cid* is *low***

$$[\mathrm{S}_{low}] \quad \frac{Exp : low}{\mathsf{send}(cid, Exp) \hookrightarrow \mathsf{send}(cid, Exp) : \mathsf{send}(cid, Exp)}$$

$$[\mathrm{R}_{low}] \quad \mathsf{receive}(cid, var) \hookrightarrow \mathsf{receive}(cid, var) : \mathsf{receive}(cid, \hat{var})$$

$$[\mathrm{IR}_{low}] \quad \frac{C_1 \hookrightarrow C_1' : Sl_1 \quad C_2 \hookrightarrow C_2' : Sl_2}{\mathsf{if\text{-}receive}(cid,var,C_1,C_2) \hookrightarrow \mathsf{if\text{-}receive}(cid,var,C_1',C_2') : \mathsf{if\text{-}receive}(cid,\hat{var},Sl_1,Sl_2)}$$

$$[\mathrm{SS}_{low}] \quad \frac{Exp : low}{\mathsf{synch\text{-}send}(cid, Exp) \hookrightarrow \mathsf{synch\text{-}send}(cid, Exp) : \mathsf{synch\text{-}send}(cid, Exp)}$$

$$[\mathrm{SR}_{low}] \quad \mathsf{synch\text{-}receive}(cid, var) \hookrightarrow \mathsf{synch\text{-}receive}(cid, var) : \mathsf{synch\text{-}receive}(cid, \hat{var})$$

**Fig. 5.** Security typing of expressions and communication primitives

[6, 44]) to equalize the timing behavior independently whether the channel $cid$ is empty (the rule [$\mathrm{IR}_{high}$]). To (conservatively) prevent potential implicit flows, no assignment to $l$ or communication on encrypted or low channels is allowed in the branches. Let $al(C)$ be a boolean function returning *true* whenever there is a syntactic occurrence of either an assignment to $l$ or a communication primitive on encrypted or low channels. The condition $al(Sl_1) = al(Sl_2) = \mathit{false}$ prevents implicit leaks. The slice of the overall command is the sequential composition of the slices of the branches prefixed with a skip corresponding to the time tick for the guard inspection.

Typing communication primitives on encrypted channels is compositional, i.e., the type (slice) of a program is assembled from the types (slices) of the program's components. Although the low slices of the typing cannot use $h$, they still must expose the same behavior wrt $|cid|$. This is achieved by using 0 instead of the potentially high *Exp* in the slice in [$\mathrm{S}_{enc}$] and [$\mathrm{SS}_{enc}$], and using versions of receive, if-receive, synch-receive that do not update $h$ in [$\mathrm{R}_{enc}$], [$\mathrm{IR}_{enc}$] and [$\mathrm{SR}_{enc}$]. For this purpose we use the notation $\hat{var}$ where $\hat{l} = l$ and $\hat{h} = \_$ to indicate the case that $h$ is not updated after a reception. Note that the generation of such slices does not change the semantics of the original program. Indeed, no low slice $Sl$ created in [$\mathrm{S}_{enc}$] or [$\mathrm{SS}_{enc}$] is used in cross-copying rules because $al(Sl) = \mathit{true}$. Directed by the hook-up result, we proceed analogously to derive the types of communication primitives on low channels.

*Correctness of the analysis.* The correctness of the analysis follows from the observation that the attacker cannot distinguish the result of the transformation and its slice. In other words, $\vec{C} \hookrightarrow \vec{C'} : \vec{Sl}$ implies $\vec{C'} \approxeq_L \vec{Sl}$, which can be proved by the hook-up theorem. By the transitivity of $\approxeq_L$ we have $\vec{C'} \approxeq_L \vec{C'}$. This leads us to the correctness theorem.

**Theorem 2 (Correctness of the Analysis)** $\vec{C} \hookrightarrow \vec{C'} : \vec{Sl} \Longrightarrow \vec{C'}$ *is secure.*

## 6 Example of Secure Programming: A File Server

The correctness theorem guarantees robust timing-sensitive security, but are there any useful secure programs? While we refer to [7] for efficient secure algorithms (such as sorting and searching) in a sequential setting, let us return to the distributed and multi-threaded file server and client programs and sketch how they can be typed.

Consider the program fragments in Figure 6. The multi-threaded file server *ServerMan* dynamically creates threads $Server[i]$ ($i \in \mathbb{N}$) upon new requests from a file client $Client[j]$ with the thread identifier $j \in \mathbb{N}$. The purpose of the file server is to store confidential data and to support fast accesses over a network that might be audited by attackers. Hence, the use of encrypted channels is a natural necessity.[5] The channels $open$, $open[i]$, $openReply[i]$, $access[i]$ are

[5] This is practiced, e.g., at the "privacy level" in the DCE/DFS distributed file system.

```
ServerMan = while true
              do receive(open, (fileName, clientID));
                 .../* generate a fresh id i for the thread to be created */
                 fork(skip, Server[i]);   /* fork off a thread handling accesses */
                 send(open[i], (fileName, clientID))
Server[i] = receive(open[i], (fname, clid));
            .../* open file fname; if successful then: */
            send(openReply[clid], i); more := true;
            while more
              do receive(access[i], request);
                 if action(request) = READ then ...
                 if action(request) = WRITE then ...
                 if action(request) = CLOSE then ... more := false
                 send(accessReply[clid], ...)   /* send access results */
Client[j] = send(open, ("foo", j)); receive(openReply[j], serverID);
            send(access[serverID], accessArgs); receive(accessReply[j], results)
```

**Fig. 6.** A multi-threaded file server and a file client

encrypted channels. Thus, assuming the variables *fileName*, *fname*, *accessArgs*, *request*, *results* are *high*, these programs can be typable and, thus, are secure.[6]

This example illustrates that our security condition and type system, although being rigorous enough to ensure security, are not over-restrictive. Namely, useful programs that fulfill our conditions can be written. Note that using encrypted channels we are able to avoid a busy-loop waiting overhead (that would have been necessary, e.g., in [33]) and yet guarantee timing-sensitive security. Due to space limitations, the example focuses on only two primitives, send and receive. However, similar examples exist for the usability of if-receive, synch-send and synch-receive wrt our security condition.

## 7 Conclusions

It might seem astonishing that, in some cases, secure programming with protected (high) channels is more restrictive than with encrypted or cleartext (low) channels. However, this is the case for the receive primitive which is insecure for high channels but secure for encrypted and low channels. This phenomenon stems from the nonmonotonic nature of confidentiality properties: the fact that

[6] Note that the example assumes an extension of our language with pairs and arrays. Security definition and the type system (along with the correctness proof) can be adjusted for these constructs following [6].

the attacker observes less about data does not imply that the data requires less (or, conversely, more) protection. Indeed, as can be seen from our security specification, as the attacker's observational power decreases, the range of possible values of sensitive data increases.

*Contributions.* In order to investigate the interplay between the observational power of the attacker and the expressive power of a programming language featuring multi-threading, (non)blocking and (a)synchronous message passing, we have proposed a timing-sensitive confidentiality specification. This specification accommodates three types of communication channels (for high, encrypted and low data) and enjoys a hook-up property wrt a number of contexts. To the best of our knowledge, this article is the first to prove hook-up properties for timing-sensitive confidentiality in a distributed language. As an important contribution, we have derived a sound security-type system from the hook-up result. Essential implications for programming secure systems are summarized in Section 4.3.

This article follows [33] as a starting point, where security in a distributed language is defined. The novelties of this article wrt [33] include an investigation of encrypted channels, synchronous communication primitives, an analysis of what primitives can be used securely for what channels, a development of hook-up properties at the level of commands[7] and, most importantly, a security enforcement mechanism in the form of a security-type system.

*Related work.* As far as we are aware, Reitman's security logic [41] is the first to address message-passing primitives in the context of confidentiality. His primitives correspond to our asynchronous send and receive with only one generic type of a communication channel. Banâtre and Bryce's security logic [10] describes security properties for a language based on CSP [24]. Their primitives correspond to our synchronous send and receive and, thus, their use is severely restricted. Both of these logics lack noninterference proofs and automatic inference algorithms. Furthermore, they do not treat timing flows. Mizuno and Oldehoeft [38] consider information flow in an object-oriented distributed system. The lack of noninterference results and a costly runtime security mechanism (that is invoked every time a message is sent) are major drawbacks of this work.

Abadi and Blanchet [1, 3] have devised type systems that guarantee confidentiality for a calculus of cryptographic protocols, the spi calculus [4]. This approach is capable of handling key-exchange protocols. However, their formalism is specific to the spi-calculus: secret keys and their usage are hidden from the low-observable view so that the resulting security condition allows for protocols with encryption to be represented in a noninterference-like fashion.

Honda et al. propose a powerful type system for tracking information flow in the asynchronous pi-calculus [25, 26]. While not designed for expressing information flow in high-level languages, their rich *channel types* allow for encoding other type systems (e.g., [47]) for high-level multi-threaded languages.

[7] The compositionality principle of [33] focuses on the level of processes rather than on the level of commands. It asserts that the composition of secure processes results in an overall system that satisfies a global trace-based security property.

Zdancewic et al. introduce *secure program partitioning* as a technique for secure distributed programming on heterogeneously trusted hosts [52]. This approach is appealing because it addresses the interaction between confidentiality and integrity in a distributed system. However, no convincing security assurance results are available for secure program partitioning.

There is a large body of work on noninterference in the setting of process algebras (see [18, 42] for an overview) and other event-based systems (see, e.g., [30] for an overview). Common ground with many process algebra-based security specifications and our study is bisimulation-based security and an asynchronous semantics of computation (which allows for modeling subcomputations with different relative speeds). However, our primary focus is different in considering (i) a concrete programming language, (ii) a timing-sensitive security specification and (iii) asynchronous message passing. Although some recent process-algebra-based investigations have explored timing-sensitive security (e.g., [19]) and security in an asynchronous setting (e.g., [23, 26]), timing is often disregarded; and handshaking models that require both sides to synchronize during communication are typically used. There is no such requirement under asynchronous message passing (as expressed by send, receive and if-receive), which is suitable for modeling programming in a distributed environment.

*Future work.* While we have assumed that programs are executed on trusted hosts, some machines may be entirely controlled by the attacker in a realistic distributed setting. Thus, an important goal for future work is to obtain a confidentiality specification that is sensitive to malicious hosts.

It is essential to lift another practically unrealistic assumption—that all messages are reliably delivered. We expect that both the security condition and the type system are robust wrt message loss. Indeed, assuming communication failures do not depend on sensitive data, we can model a lossy channel *cid* by the process $\mathit{fail}(\mathit{cid}) = \mathsf{while}\ \mathit{true}\ \mathsf{do}\ \mathsf{if\text{-}receive}(\mathit{cid}, x_{\mathit{cid}}, \mathsf{skip}, \mathsf{skip})$ for a fresh high variable $x_{\mathit{cid}}$. That the security of a system cannot be compromised by such a failure process is a simple corollary of the hook-up result and the observation that $\mathit{fail}(\mathit{cid})$ is secure for any *cid*.

**Acknowledgments** Thanks to Dan Grossman, Andrew Myers, David Sands, Fred Schneider, Peter Sewell, and Steve Zdancewic for insightful feedback.

## References

1. M. Abadi. Secrecy by typing in security protocols. *Journal of the ACM*, 46(5):749–786, September 1999.
2. M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. A core calculus of dependency. In *Proc. of Symposium on Principles of Programming Languages*, January 1999.
3. M. Abadi and B. Blanchet. Secrecy types for asymmetric communication. In *FOSSACS'01*, *LNCS 2030*, pages 25–41, April 2001.
4. M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *Proc. of Conference on Computer and Communications Security*, 1997.

5. M. Abadi and P. Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). *Journal of Cryptology*, 15(2), 2002.
6. J. Agat. Transforming out timing leaks. In *Proc. of Symposium on Principles of Programming Languages*, pages 40–53, January 2000.
7. J. Agat and D. Sands. On confidentiality and algorithms. In *Proc. of IEEE Symposium on Security and Privacy*, May 2001.
8. G. R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison Wesley, 2000.
9. J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice-Hall, 2nd edition, 1996.
10. J.-P. Banâtre and C. Bryce. Information flow control in a parallel language framework. In *Proc. of IEEE Computer Security Foundations Workshop*, June 1993.
11. A. Banerjee and D. A. Naumann. Secure information flow and pointer confinement in a Java-like language. In *Proc. of IEEE Computer Security Foundations Workshop*, June 2002.
12. G. Boudol and I. Castellani. Noninterference for concurrent programs. In *ICALP'01*, *LNCS 2076*, pages 382–395, July 2001.
13. E. S. Cohen. Information transmission in computational systems. *ACM SIGOPS Operating Systems Review*, 11(5):133–139, 1977.
14. E. S. Cohen. Information transmission in sequential programs. In R. A. DeMillo, D. P. Dobkin, A. K. Jones, and R. J. Lipton, editors, *Foundations of Secure Computation*, pages 297–335. Academic Press, 1978.
15. D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.
16. D. E. Denning. *Cryptography and Data Security*. Addison-Wesley, 1982.
17. D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.
18. R. Focardi and R. Gorrieri. A classification of security properties for process algebras. *Journal of Computer Security*, 3(1):5–33, 1995.
19. R. Focardi, R. Gorrieri, and F. Martinelli. Information flow analysis in a discrete-time process algebra. In *Proc. of IEEE Computer Security Foundations Workshop*, pages 170–184, July 3–5 2000.
20. J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. of IEEE Symposium on Security and Privacy*. April 1982.
21. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.
22. N. Heintze and J. G. Riecke. The SLam calculus: programming with secrecy and integrity. In *Proc. of Symposium on Principles of Programming Languages*, 1998.
23. M. Hennessy and J. Riely. Information flow vs resource access in the asynchronous pi-calculus (extended abstract). In *ICALP'00*, *LNCS 1853*, pages 415–427, 2000.
24. C. Hoare. *Communicating Sequential Processes (CSP)*. Prentice Hall, 1985.
25. K. Honda, V. Vasconcelos, and N. Yoshida. Secure information flow as typed process behaviour. In *Proc. of European Symposium on Programming*, *LNCS 1782*, pages 180–199, 2000.
26. K. Honda and N. Yoshida. A uniform type structure for secure information flow. In *Proc. of Symposium on Principles of Programming Languages*, January 2002.
27. B. W. Lampson. A note on the confinement problem. *Commun. ACM*, 16(10):613–615, Oct. 1973.
28. P. Laud. Semantics and program analysis of computationally secure information flow. In *Proc. of European Symposium on Programming*, *LNCS 2028*, April 2001.

29. K. R. M. Leino and R. Joshi. A semantic approach to secure information flow. *Science of Computer Programming*, 37(1–3):113–138, 2000.
30. H. Mantel. Possibilistic definitions of security – an assembly kit –. In *Proc. of IEEE Computer Security Foundations Workshop*, pages 185–199, July 2000.
31. H. Mantel. On the composition of secure systems. In *Proc. of IEEE Symposium on Security and Privacy*, May 2002.
32. H. Mantel and A. Sabelfeld. A generic approach to the security of multi-threaded programs. In *Proc. of IEEE Computer Security Foundations Workshop*, June 2001.
33. H. Mantel and A. Sabelfeld. A unifying approach to the security of distributed and multi-threaded programs. *Journal of Computer Security*, 2002. To appear.
34. D. McCullough. Specifications for multi-level security and hook-up property. In *Proc. of IEEE Symposium on Security and Privacy*, pages 161–166. May 1987.
35. J. McLean. The specification and modeling of computer security. *Computer*, 23(1), January 1990.
36. J. McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Proc. of IEEE Symposium on Security and Privacy*, 1994.
37. R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.
38. M. Mizuno and A. Oldehoeft. Information flow control in a distributed object-oriented system with statically-bound object variables. In *Proc. of National Computer Security Conference*, pages 56–67, 1987.
39. B. Pfitzmann and M. Waidner. A model for asynchronous reactive systems and its application to secure message transmission. In *Proc. of IEEE Symposium on Security and Privacy*, pages 184–200, 2001.
40. A. D. Pierro, C. Hankin, and H. Wiklicky. Approximate non-interference. In *Proc. of IEEE Computer Security Foundations Workshop*, June 2002.
41. R. P. Reitman. *Information flow in parallel programs: an axiomatic approach.* PhD thesis, Cornell University, 1978.
42. P. Ryan, S. Schneider, M. Goldsmith, G. Lowe, and B. Roscoe. *The Modelling and Analysis of Security Protocols: the CSP Approach.* Addison-Wesley, 2001.
43. A. Sabelfeld. The impact of synchronisation on secure information flow in concurrent programs. In *Proc. of International Conference on Perspectives of System Informatics, LNCS 2244*, pages 227–241, July 2001.
44. A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Proc. of IEEE Computer Security Foundations Workshop*, July 2000.
45. A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. *Higher-Order and Symbolic Computation*, 14(1):59–91, 2001.
46. G. Smith. A new type system for secure information flow. In *Proc. of IEEE Computer Security Foundations Workshop*, pages 115–125, June 2001.
47. G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. of Symposium on Principles of Programming Languages*, 1998.
48. D. Volpano. Safety versus secrecy. In *Proc. of Symposium on Static Analysis, LNCS 1694*, pages 303–311, September 1999.
49. D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. *Journal of Computer Security*, 7(2,3):231–253, November 1999.
50. D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.
51. S. Zdancewic and A. C. Myers. Secure information flow via linear continuations. *Higher Order and Symbolic Computation*, 2002. To appear.
52. S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Untrusted hosts and confidentiality: Secure program partitioning. In *Proc. of ACM Symposium on Operating System Principles*, October 2001.

## Appendix A

<table>
<tr><td>✓="secure" ×="insecure"</td><td colspan="2">$dom(cid)=dom(cid')$</td></tr>
<tr><th>program</th><th>cid- enc</th><th>cid- high</th></tr>
<tr><td colspan="3">send:</td></tr>
<tr><td>send($cid, Exp$)</td><td colspan="2">✓</td></tr>
<tr><td>if $Exp_h$ then send($cid, Exp$) else skip</td><td>×</td><td>✓</td></tr>
<tr><td>if $Exp_h$ then send($cid, Exp$) else send($cid', Exp'$)</td><td>×</td><td>✓</td></tr>
<tr><td>if $Exp_h$ then send($cid, Exp$) else send($cid, Exp'$)</td><td colspan="2">✓</td></tr>
<tr><td>send($cl, Exp_h$) ($dom(cl) = low$)</td><td colspan="2">×</td></tr>
<tr><td colspan="3">receive:</td></tr>
<tr><td>receive($cid, h$)</td><td>✓</td><td>×</td></tr>
<tr><td>if $Exp_h$ then receive($cid, h$) else skip</td><td colspan="2">×</td></tr>
<tr><td>if $Exp_h$ then receive($cid, h$) else receive($cid', h$)</td><td colspan="2">×</td></tr>
<tr><td>if $Exp_h$ then receive($cid, h$) else receive($cid, h'$)</td><td>✓</td><td>×</td></tr>
<tr><td>receive($cid, l$)</td><td colspan="2">×</td></tr>
<tr><td colspan="3">if-receive:</td></tr>
<tr><td>if-receive($cid, h, C_1, C_2$)</td><td>$C_1 \approxeq_L C_1$<br>$C_2 \approxeq_L C_2$</td><td>$C_1 \approxeq_L C_2$</td></tr>
<tr><td>if $Exp_h$ then if-receive($cid, h, C_1, C_2$) else (skip; $C_3$)</td><td>×</td><td>$C_1 \approxeq_L C_2$<br>$C_2 \approxeq_L C_3$</td></tr>
<tr><td>if-receive($cid, l, C_1, C_2$)</td><td colspan="2">×</td></tr>
<tr><td colspan="3">synch-send ($dom(destReady[cid]) = dom(destAck[cid]) = dom(cid)$ for $cid \in CID$):</td></tr>
<tr><td>synch-send($cid, h$)</td><td>✓</td><td>×</td></tr>
<tr><td>if $Exp_h$ then synch-send($cid, Exp$) else skip</td><td colspan="2">×</td></tr>
<tr><td>if $Exp_h$ then synch-send($cid,Exp$) else synch-send($cid',Exp'$)</td><td colspan="2">×</td></tr>
<tr><td>if $Exp_h$ then synch-send($cid,Exp$) else synch-send($cid,Exp'$)</td><td>✓</td><td>×</td></tr>
<tr><td>synch-send($cl, Exp_h$) ($dom(cl) = low$)</td><td colspan="2">×</td></tr>
<tr><td>if $Exp_h$ then send($cid, Exp$) else synch-send($cid, Exp$)</td><td colspan="2">×</td></tr>
<tr><td colspan="3">synch-receive ($dom(destReady[cid]) = dom(destAck[cid]) = dom(cid)$ for $cid \in CID$):</td></tr>
<tr><td>synch-receive($cid, h$)</td><td>✓</td><td>×</td></tr>
<tr><td>if $Exp_h$ then synch-receive($cid, h$) else skip</td><td colspan="2">×</td></tr>
<tr><td>if $Exp_h$ then synch-receive($cid, h$) else synch-receive($cid', h$)</td><td colspan="2">×</td></tr>
<tr><td>if $Exp_h$ then synch-receive($cid, h$) else synch-receive($cid, h'$)</td><td>✓</td><td>×</td></tr>
<tr><td>synch-receive($cid, l$)</td><td colspan="2">×</td></tr>
<tr><td>if $Exp_h$ then receive($cid, h$) else synch-receive($cid, h$)</td><td colspan="2">×</td></tr>
</table>

**Fig. 7.** Examples on communication primitives and their impact on security