Taming Message-passing Communication in Compositional Reasoning about Confidentiality

Ximeng Li Heiko Mantel Markus Tasch

Department of Computer Science, TU-Darmstadt, Germany
{li, mantel, tasch}@mais.informatik.tu-darmstadt.de

Abstract. We propose a solution for verifying the information-flow security of distributed programs in a compositional manner. Our focus is on the treatment of message passing in such a verification, and our goal is to boost the precision of modular reasoning using rely-guarantee-style reasoning. Enabling a more precise treatment of message passing required the identification of novel concepts that capture assumptions about how a process's environment interacts. Our technical contributions include a process-local security condition that allows one to exploit such assumptions when analyzing individual processes, a security type system that is sensitive in the content as well as in the availability of messages, and a soundness proof for our security type system. Our results complement existing solutions for rely-guarantee-style reasoning about information-flow security that focused on multi-threading and shared memory.

1 Introduction

Information-flow security aims at end-to-end security guarantees for IT systems. Noninterference [9] is a formally defined property that requires a system's publicly observable output to not depend on secrets. Multiple variants of noninterference exist and can be used to capture that system runs do not reveal information about secrets [17]. The formal verification of a noninterference-like property provides a high level of assurance, but can be non-trivial.

Compositional reasoning can be applied to reduce the conceptual complexity of a formal verification of information-flow security [21]. In compositional reasoning, one infers the security of a complex system from the security of its individual components based on a compositionality theorem. The local security of an individual system component can, again, be expressed using a noninterference-like property. To ensure soundness of compositional reasoning, the locally verified security properties might be more restrictive than the globally desired security guarantees. Hence, a key question in compositional reasoning is how to benefit from the reduction of conceptual complexity without loosing too much precision.

One can increase the precision of modular reasoning by exploiting assumptions in the local verification of a component, i.e., by rely-guarantee reasoning [13]. While being established in other areas, sound solutions for rely-guarantee-style reasoning about information-flow security appeared relatively late [19]. Moreover, their focus was on systems with shared-memory communication [3,18,

To appear in:
Bor-Yuh Evan Chang (Eds.): APLAS 2017

19, 22]. This article is complementary in this respect. We develop a solution for rely-guarantee-style reasoning for systems with message-passing communication.

The problem of verifying information-flow security for systems with message passing in a modular fashion has been addressed before, e.g., in [7,25]. There are two main reasons why message passing received special attention. Firstly, the fact that a message is present or not might communicate information that needs to be protected. Hence, it does not suffice to protect just the confidentiality of a message's content [25]. Secondly, messages are usually buffered until they have been received. Hence, when a message is received, it is deleted from a buffer [27]. That is, receive commands differ from read commands in that they block if no message is present and in that they alter a system's state when a message is received. Such peculiarities of message passing must be taken into account.

A solution for rely-guarantee-style reasoning about information-flow security for systems with remote procedure calls was proposed in [10]. This constitutes a step towards supporting message-passing communication, but it is limited to systems with unbufferred communication and with a hierarchical communication structure. In this article, we aim for a general solution that, in particular, addresses buffered message-passing communication and that is suitable for the flexible interaction patterns commonly used in distributed systems. Ultimately, the goal is to support both fundamental principles of communication (i.e, message passing and shared memory [27]). The solution for message passing presented here is compatible with the solution for shared memory in [19].

The main technical contributions of this article are:

(A) a sound, yet precise solution for modular reasoning about distributed system security at the semantic level, via a novel process-local security condition
(B) a sound, fine-grained security analysis that provides the basis for automating the verification of the process-local security condition

More concretely, our process-local security condition, called local security, captures the intuition that an individual process of a distributed system does not leak secrets under explicit assumptions about the process's interface. The ability to exploit assumptions increases the precision of local reasoning about security. Local security is compositional in allowing one to conclude the security of the overall distributed system from the local security of all processes. This compositionality lifts the precision gain in local reasoning to the global level.

Our analysis for local security is realized as a security type system [28]. This security type system exploits the assumptions explicitly made in a program to perform fine-grained type checking of the program. Typing constraints are relaxed for a receive command when an assumption indicates the non-blockage of the receive. Furthermore, the type system is content-sensitive and availability-sensitive: it deals with programs where the confidentiality of the content and presence of messages may vary over the same communication channel.

We evaluate our approach at examples with communication patterns taken from real-world distributed systems. All processes in the examples are typable, and the systems are proven secure using the compositionality of local security.

2 Motivation and Approach at a Glance

We illustrate that without making assumptions about message passing, the local information-flow analysis of processes is overly restrictive. For three simplistic example programs, we provide environments in which the programs cause information leakage and environments in which the programs behave securely. That is, these programs must be classified as insecure unless one is able to characterize the environments in which the respective program behaves securely.

We propose a language for specifying assumptions about a component's environment. We explain how one can apply this language to capture environments in which our three example programs behave securely. The identification of assumptions that are suitable for this purpose was a key step in our research project. By exploiting the assumptions in a compositional verification of information-flow security, one can increase the precision of the overall analysis substantially.

Consider the following three example programs:

$$\begin{aligned} \mathit{content} &= \mathsf{recv}(\underline{\mathit{enc}}, x); \mathsf{send}(\underline{\mathit{pub}}, x) \\ \mathit{presence}_1 &= \mathsf{recv}(\underline{\mathit{pri}}, x); \mathsf{send}(\underline{\mathit{pub}}, 1) \\ \mathit{presence}_2 &= \mathsf{if}\ h > 0\ \mathsf{then}\ \mathsf{recv}(\underline{\mathit{pri}}, x)\ \mathsf{else}\ \mathsf{skip}\ \mathsf{fi}; \mathsf{send}(\underline{\mathit{pub}}, 1) \end{aligned}$$

where x is a *local variable* (i.e., it cannot be accessed by other processes), h is a *high local variable* (i.e., it might contain a secret), $\underline{\mathit{pri}}$ is a *private channel* (i.e., its content cannot be seen by attackers), $\underline{\mathit{enc}}$ is an *encrypted channel* (i.e., the existence of messages might be visible to attackers, but not the messages' content), and $\underline{\mathit{pub}}$ is a *public channel* (i.e., its content might be visible to attackers).

That is, *content* awaits a message via an encrypted channel and forwards this message over a public channel. The program $\mathit{presence}_1$ awaits a message from a private channel, and then outputs 1 to a public channel. If its then-branch is taken, $\mathit{presence}_2$ awaits a message from a private channel and subsequently outputs 1 to a public channel. Otherwise, $\mathit{presence}_2$ just outputs 1 to the public channel. Note that which branch is chosen depends on potentially secret information. There is a danger of information leakage for each of these programs, but there are also environments in which the programs behave securely.

Example 1. If an environment sends a secret over the encrypted channel $\underline{\mathit{enc}}$ then *content* causes information leakage because it forwards the secret to the public channel $\underline{\mathit{pub}}$, whose content might be visible to attackers. In contrast, if the environment provides a public message over $\underline{\mathit{enc}}$ (e.g., a username over HTTPS) then *content* forwards a public value to $\underline{\mathit{pub}}$, i.e., no secrets are leaked.

If an environment only sends a message over $\underline{\mathit{pri}}$ if some secret value is positive then $\mathit{presence}_1$ leaks information: The attacker sees 1 on $\underline{\mathit{pub}}$ only if the secret is positive. Otherwise, $\underline{\mathit{pri}}$ blocks at the receive command and the send command is never reached. However, if secrets do not influence whether the environment sends messages on $\underline{\mathit{pri}}$ then $\mathit{presence}_1$ does not cause information leakage.

If an environment does not send any messages over $\underline{\mathit{pri}}$ then $\mathit{presence}_2$ causes information leakage: The attacker sees 1 on $\underline{\mathit{pub}}$ only if the value of h is negative or zero. If the value of h is positive, then $\mathit{presence}_2$ blocks at the receive command and the send command is never reached. In contrast, if the environment definitely

sends some message over $\underline{pri}$ before the command $\mathsf{recv}(\underline{pri}, x)$ is reached then the program $presence_2$ does not cause any information leakage. $\diamond$

Our Solution at a Glance. The program *content* does not leak secrets in environments that provide a public message on $\underline{enc}$ as explained in Example 1. We explicate that we expect the next available message to have public content by annotating the receive command in *content* with $\mathbb{L}^\bullet$, which results in:

$$^{\{\mathbb{L}^\bullet\}}\mathsf{recv}(\underline{enc}, x); \mathsf{send}(\underline{pub}, x)$$

The annotation $\mathbb{L}^\bullet$ indicates that the program shall only be run in environments that guarantee a public message to be received. This assumption can then be exploited to make the local security analysis more permissive.

For explicating that $presence_1$ and $presence_2$ shall only run in environments that provide messages on $\underline{pri}$ before the programs try to receive, we employ NE:

$$^{\{NE\}}\mathsf{recv}(\underline{pri}, x); \mathsf{send}(\underline{pub},1)$$
$$\mathsf{if}\ h > 0\ \mathsf{then}\ ^{\{NE\}}\mathsf{recv}(\underline{pri}, x)\ \mathsf{else\ skip\ fi}; \mathsf{send}(\underline{pub}, 1)$$

We introduce $\mathbb{L}^\circ$ to capture that the availability of messages does not depend on secrets. This assumption is weaker than the one captured by NE. Since $presence_1$ does not leak secrets in environments satisfying $\mathbb{L}^\circ$, we can use it instead of NE:

$$^{\{\mathbb{L}^\circ\}}\mathsf{recv}(\underline{pri}, x); \mathsf{send}(\underline{pub},1)$$

In summary, we propose the following language of assumptions:

$$asm ::= \mathbb{L}^\bullet \mid NE \mid \mathbb{L}^\circ$$

These assumptions can be used to constrain the environments in which a program may be run. We will show how to exploit the assumptions in modular reasoning about information-flow security, both at a semantic level (Sect. 5) and at a syntactic level (Sect. 6). Note that our annotations are helpful hints for the security analysis, but they do not alter a program's behavior in any way.

3 Model of Computation

We consider distributed systems whose processes have a local memory and communicate using message passing both with each other and with the environment. We model the communication lines used for message passing by a set of *channels* Ch. We use $ECh \subseteq Ch$ to denote the set of *external channels*, i.e. the system's interface to its environment. We use $ICh = Ch \setminus ECh$ to denote the set of *internal channels*, i.e. the channels that cannot be accessed by the system's environment.

Snapshots. We model each snapshot of a single process by a pair $\langle prog; mem\rangle$ consisting of a control state *prog* and a memory state *mem*. The *control state prog* equals the program that remains to be executed by the process. We denote the set of all programs by *Prog* and leave this set underspecified. The *memory state mem* is a function of type $Mem = Var \to Val$, where Var is a set of *program variables* and Val is a set of *values*. We call pairs $\langle prog; mem\rangle$ *process configurations* and use $PCnf$ to denote the set of all process configurations.

We model each snapshot of a distributed system by a pair $\langle pcnfl; \sigma \rangle$ consisting of a list of process configurations $pcnfl \in PCnf^*$ and a channel state σ. The list $pcnfl$ models the local state of each process. The *channel state* σ is a function of type $\Sigma = Ch \to Val^*$ that models, by a FIFO queue, which messages have been sent, but not yet received on each communication line.

We call pairs $\langle pcnfl; \sigma \rangle$ *global configurations* and use $GCnf$ to denote the set of all global configurations.

We use *local configurations* to capture the local view of individual processes during a run. Formally, a local configuration $lcnf$ is a pair $\langle pcnf; \sigma \rangle$ where $pcnf \in PCnf$ and $\sigma \in \Sigma$. In a global configuration $\langle [pcnf_1, \ldots, pcnf_i, \ldots, pcnf_n]; \sigma \rangle$, the local configuration of process i is $\langle pcnf_i; \sigma \rangle$.

Runs. We model runs of distributed systems by *traces*, i.e. finite lists of global configurations, and use $Tr = GCnf^*$ to denote the set of all traces. Appending a global configuration to the end of a trace captures either the effect of a computation step by the distributed system or of an interaction with its environment.

We model system environments by functions of type $\Xi = Tr \to (ECh \to Val^*)$, which we call *strategies*. The function $\xi(tr)$ defines which inputs the environment modeled by ξ provides and which outputs it consumes on each external communication line after the run modeled by tr. That is, $\xi(tr)(ch)$ equals the content of $ch \in ECh$ after the environment's interaction subsequent to tr.

We use the judgment $tr \to_\xi tr'$ to capture that the trace tr' might result by one computation or communication step after the trace tr under the strategy ξ. The calculus for this judgment consists of the following two rules.

$$\frac{\langle pcnfl; \sigma \rangle = last(tr) \quad \sigma' = \xi(tr)}{tr \to_\xi tr \cdot \langle pcnfl; \sigma[ECh \mapsto \sigma'(ECh)] \rangle} \qquad \frac{\langle [\ldots, pcnf_i, \ldots]; \sigma \rangle = last(tr) \quad \langle pcnf_i; \sigma \rangle \to \langle pcnf_i'; \sigma' \rangle}{tr \to_\xi tr \cdot \langle [\ldots, pcnf_i', \ldots]; \sigma' \rangle}$$

The first rule captures how the environment interacts with the system using external communication lines. The expression $\sigma[ECh \mapsto \sigma'(ECh)]$ in the conclusion of this rule denotes the pointwise update of the function σ such that each $ch \in ECh$ is mapped to $\sigma'(ch)$ and each $ch \in ICh$ to $\sigma(ch)$. Note that one communication step might update multiple channels.

The second rule captures how the system itself performs a step. The choice of the acting process is non-deterministic, which reflects a distributed system without global scheduler. The judgment $\langle pcnf_i; \sigma \rangle \to \langle pcnf_i'; \sigma' \rangle$ (second premise of the rule) captures how the chosen process updates its own configuration and the channel state. We will introduce a calculus for this judgment later.

We model the possible runs of a distributed system by a set of traces. We define this set by $traces_\xi(gcnf) = \{tr' \in Tr \mid [gcnf]\ (\to_\xi)^*\ tr'\}$, where $gcnf \in GCnf$ is an initial configuration and $(\to_\xi)^*$ the reflexive transitive closure of $\to_\xi$.

We view a *distributed program* as a list of programs, use $\|$ to indicate that the programs in the list are executed by concurrently running processes, and define the set of all distributed programs by $DProg = Prog^*$. Given a distributed program $dprog = [prog_1 \parallel \cdots \parallel prog_n]$, we define its possible traces under ξ by

$$traces_\xi(dprog) = traces_\xi(\langle [\langle prog_1; mem_{\text{init}} \rangle, \ldots, \langle prog_n; mem_{\text{init}} \rangle]; \sigma_{\text{init}} \rangle)$$

The initial memory mem_{init} maps all variables to the designated initial value v_{init}, and the initial channel state σ_{init} maps all channels to the empty list ϵ.

Programming Language. We consider a simple while-language extended with message-passing commands to make things concrete. The syntax is defined by

$$\begin{aligned} \mathit{prog} ::= {} & \mathsf{recv}(\mathit{ch}, x) \mid \mathsf{if\text{-}recv}(\mathit{ch}, x, x_{\mathrm{b}}) \mid \mathsf{send}(\mathit{ch}, e) \mid \mathsf{skip} \mid x := e \mid \\ & \mathit{prog}_1; \mathit{prog}_2 \mid \mathsf{if}\ e\ \mathsf{then}\ \mathit{prog}_1\ \mathsf{else}\ \mathit{prog}_2\ \mathsf{fi} \mid \mathsf{while}\ e\ \mathsf{do}\ \mathit{prog}\ \mathsf{od} \mid \mathsf{stop} \end{aligned}$$

Derivability of the judgment $\langle \mathit{pcnf}; \sigma \rangle \to \langle \mathit{pcnf}'; \sigma' \rangle$ is defined by a structural operational semantics of the programming language, where rules for most commands are straightforward (and, hence, omitted here to save space).

The command skip terminates without effects. An assignment $x := e$ stores the value of the expression e in x. Sequential composition $\mathit{prog}_1; \mathit{prog}_2$, conditional branching $\mathsf{if}\ e\ \mathsf{then}\ \mathit{prog}_1\ \mathsf{else}\ \mathit{prog}_2\ \mathsf{fi}$, and loops $\mathsf{while}\ e\ \mathsf{do}\ \mathit{prog}\ \mathsf{od}$ have the usual semantics. The symbol stop is used to signal that the process has terminated, and it is not part of the surface syntax used for writing programs.

The blocking receive $\mathsf{recv}(\mathit{ch}, x)$ removes the first message from the message queue $\sigma(\mathit{ch})$ and stores it in x (first rule on the right). If no message is available on ch, then the execution blocks until a message becomes available. The non-blocking receive $\mathsf{if\text{-}recv}(\mathit{ch}, x, x_{\mathrm{b}})$ uses the variable x_{b} to record whether a receive was successful ($x_{\mathrm{b}} = \mathit{tt}$) or not ($x_{\mathrm{b}} = \mathit{ff}$). If no message could be received, then the program run continues with x unchanged. The send command appends the value of e to $\sigma(\mathit{ch})$ (second rule above).

$$\frac{\sigma(\mathit{ch}) = v \cdot \gamma \quad \mathit{mem}' = \mathit{mem}[x \mapsto v] \quad \sigma' = \sigma[\mathit{ch} \mapsto \gamma]}{\langle\langle \mathsf{recv}(\mathit{ch}, x); \mathit{mem}\rangle; \sigma\rangle \to \langle\langle \mathsf{stop}; \mathit{mem}'\rangle; \sigma'\rangle}$$

$$\frac{[\![e]\!]_{\mathit{mem}} = v \quad \sigma' = \sigma[\mathit{ch} \mapsto \sigma(\mathit{ch}) \cdot v]}{\langle\langle \mathsf{send}(\mathit{ch}, e); \mathit{mem}\rangle; \sigma\rangle \to \langle\langle \mathsf{stop}; \mathit{mem}\rangle; \sigma'\rangle}$$

Relations between Channel States. If two channel states agree on the availability of messages on some channel, then executing a receive command on this channel either succeeds or blocks in both channel states. For each $\mathit{ch} \in \mathit{Ch}$, we introduce an equivalence relation $\overset{\mathit{ch}}{=}_{\#} \subseteq \Sigma \times \Sigma$ to characterize agreement in this respect. That is, we define $\overset{\mathit{ch}}{=}_{\#}$ by $\sigma_1 \overset{\mathit{ch}}{=}_{\#} \sigma_2$ iff $|\sigma_1(\mathit{ch})| > 0 \Leftrightarrow |\sigma_2(\mathit{ch})| > 0$.

For each $\mathit{ch} \in \mathit{Ch}$, we define a second equivalence relation $\overset{\mathit{ch}}{=}_1$ on channel states in which ch is non-empty by $\sigma_1 \overset{\mathit{ch}}{=}_1 \sigma_2$ iff $\mathit{first}(\sigma_1(\mathit{ch})) = \mathit{first}(\sigma_2(\mathit{ch}))$. We lift $\overset{\mathit{ch}}{=}_1$ to arbitrary channel states by defining channel states with $|\sigma(\mathit{ch})| = 0$ to be related to all channel states. That is, $\overset{\mathit{ch}}{=}_1 \subseteq \Sigma \times \Sigma$ is defined by $\sigma_1 \overset{\mathit{ch}}{=}_1 \sigma_2$ iff $(|\sigma_1(\mathit{ch})| > 0 \wedge |\sigma_2(\mathit{ch})| > 0) \Rightarrow \mathit{first}(\sigma_1(\mathit{ch})) = \mathit{first}(\sigma_2(\mathit{ch}))$. Note that $\overset{\mathit{ch}}{=}_1$ is intransitive and, hence, not an equivalence relation (on arbitrary channel states).

We will use $\overset{\mathit{ch}}{=}_{\#}$ and $\overset{\mathit{ch}}{=}_1$ in our definition of local security in Sect. 5.2.

Remark 1. Note that an alternative way of lifting $\overset{\mathit{ch}}{=}_1$ to arbitrary channel states is to require a channel state σ with $|\sigma(\mathit{ch})| = 0$ to be related to no other channel state. This alternative would impose the restriction that the lifted $\overset{\mathit{ch}}{=}_1$ is included in $\overset{\mathit{ch}}{=}_{\#}$. With our design choice, this restriction is not imposed, and the relations $\overset{\mathit{ch}}{=}_{\#}$ and $\overset{\mathit{ch}}{=}_1$ are kept more independent of each other.

4 Attacker Model and Baseline Security

In information-flow security, one considers an attacker who has access to parts of a system's interface and who knows how the system operates in principle (e.g., by knowing a program's code). The attacker combines the observations that he makes during a system run with his knowledge of the system logic to infer additional information. A system has secure information flow if such an attacker is unable to obtain any secrets (i.e., by his observations and by his inference).

Observation Model. As described in Sect. 3, we consider distributed systems and use the set of external channels ECh to model a system's interface. We assume that an attacker's observations are limited by access control and encryption. That is, some channels are private (messages on these channels are inaccessible), some are encrypted (the content of messages on theses channels is protected), and the remaining ones are public (the messages on these channels are unprotected). In line with Sect. 2, we model the three kinds of external channels by partitioning the set ECh into three subsets: the public channels $PubCh$, the encrypted channels $EncCh$, and the private channels $PriCh$.

If a message is transmitted over a channel, then the attacker observes the whole message – if the channel is public, its presence – if the channel is encrypted, and nothing – if the channel is private. Formally, we use a value $v \in Val$ to capture the observation of a message's content on a public channel, and use the special symbol $\odot$ to capture the observation of a message's presence on an encrypted channel. We use a sequence of values (resp. $\odot$) to capture the observation on a public (resp. encrypted) channel. Finally, we use the special symbol $\circledcirc$ to indicate the absence of an observation. That is, the set of possible observations by the attacker is $Obs = Val^* \cup \{\odot\}^* \cup \{\circledcirc\}$. We define the function $ob : (ECh \times \Sigma) \to Obs$ to retrieve the observation on an external channel from a channel state.

$$ob(ch,\sigma) \triangleq \begin{cases} \circledcirc & \text{if } ch \in PriCh \\ \odot^{|\sigma(ch)|} & \text{if } ch \in EncCh \\ \sigma(ch) & \text{if } ch \in PubCh \end{cases}$$

Attacker's Knowledge. We call two channel states distinguishable if the attacker's observations of them differ. Complementarily, we define two channel states to be *indistinguishable* for the attacker by $\sigma_1 \simeq_\Sigma \sigma_2$ iff $\forall ch \in ECh : ob(ch,\sigma_1) = ob(ch,\sigma_2)$. We call traces distinguishable if the corresponding channel states in the traces are distinguishable. Complementarily, we define two traces to be *indistinguishable* for the attacker by $tr_1 \simeq_{Tr} tr_2$ iff $\overline{ob}(tr_1) = \overline{ob}(tr_2)$ where $\overline{ob}([\langle pcnfl_1; \sigma_1\rangle, ..., \langle pcnfl_n; \sigma_n\rangle]) \triangleq [\,\lambda ch : ECh.\, ob(ch,\sigma_1), ..., \lambda ch : ECh.\, ob(ch,\sigma_n)\,]$.

The attacker's ability to distinguish different traces allows him to infer possible environments in which the system is run. More concretely, he only deems environments that are compatible with his observations possible. We capture which environments the attacker deems possible for his observations by a set of strategies and call this set the *attacker's knowledge*. We define the attacker's knowledge for each observation by the function $\mathcal{K} : (DProg \times (ECh \to Obs)^*) \to 2^\Xi$.

$$\mathcal{K}(dprog, otr) \triangleq \{\xi \in \Xi \mid \exists tr \in traces_\xi(dprog) :\ \overline{ob}(tr) = otr\}$$

Knowledge-based Security. The attacker tries to infer information about secrets coming from the environment. That is, he tries to exclude environments with the same observable interaction but different unobservable interaction. We capture that two environments differ only in their unobservable interaction by the indistinguishability relation $\simeq_\Xi$ on strategies.

$$\xi_1 \simeq_\Xi \xi_2 \triangleq \forall tr_1, tr_2 \colon tr_1 \simeq_{Tr} tr_2 \Rightarrow \forall ch \in ECh \colon ob(ch, \xi_1(tr_1)) = ob(ch, \xi_2(tr_2))$$

Based on this notion of indistinguishability on strategies, we define a distributed program to be *knowledge-based secure* (in the spirit of [4]):

Definition 1 (Knowledge-based Security). *A distributed program dprog is knowledge-based secure, denoted by KBSec(dprog), if for all $\xi \in \Xi$ and $tr \in traces_\xi(dprog)$, we have $\mathcal{K}(dprog, \overline{ob}(tr)) \supseteq \{\xi' \in \Xi \mid \xi' \simeq_\Xi \xi\}$.*

That is, a distributed program *dprog* is secure if the attacker is not able to use an observation $\overline{ob}(tr)$ made when *dprog* is run under a strategy ξ to exclude any strategy ξ' indistinguishable to ξ. Hence, the attacker is unable to exclude environments with the same observable interaction but different unobservable interaction. Thus, the attacker is unable to obtain secrets coming from the environment – *dprog* has secure information flow.

5 Compositional Reasoning about Noninterference

It is non-trivial to monolithically verify noninterference for complex systems. Our goal is to simplify such a verification by providing better support for modular reasoning. We define a notion of local security for individual processes that is a variant of noninterference (i.e., it requires the observable behavior of a process to not depend on secrets). Local security is compositional in the sense that security of a system (as defined in Sect. 4) follows from local security of all processes.

Technically, we define local security using a notion of bisimilarity on process configurations. We define a program to satisfy local security if all pairings of the program with two low-indistinguishable memories (e.g., [19,25]) are bisimilar. In the definition of this bisimilarity, we universally quantify over pairs of channel states that are indistinguishable for the attacker (as in [7, 25]). Unlike in prior work, we restrict the universal quantification of pairs of channel states further based on assumptions. This exploitation of assumptions is the key to achieving substantially better precision in a modular security analysis.

5.1 Annotated Programs

Recall our set $Asm = \{NE, \mathbb{L}^\bullet, \mathbb{L}^\circ\}$ of assumptions from Sect. 2. In order to exploit these assumptions in local reasoning, they need to be extractable from programs with annotations. We use the function *asm-of* : $Prog \to 2^{AsmCh}$ to extract which assumptions are made over which channels, from annotated programs. Here, $AsmCh$ is the set $Asm \times Ch$ of pairs of assumptions and channels.

For concrete examples, we augment our language from Sect. 3 with annotations over receive commands (abusing *prog* for programs in this new language):

$$\begin{aligned} prog ::= {} & p \mid {}^{as}\mathsf{recv}(ch, x) \mid {}^{as}\mathsf{if\text{-}recv}(ch, x, x_\mathrm{b}) \\ & p \in Prog \setminus \{\mathsf{recv}(ch, x), \mathsf{if\text{-}recv}(ch, x, x_\mathrm{b})\},\ as \subseteq Asm \end{aligned}$$

For this language, we define the function *asm-of* by $\mathit{asm\text{-}of}({}^{as}\mathsf{recv}(ch,x)) = \mathit{asm\text{-}of}({}^{as}\mathsf{if\text{-}recv}(ch,x,x_\mathrm{b})) = as \times \{ch\}$, $\mathit{asm\text{-}of}(\mathit{prog}_1;\mathit{prog}_2) = \mathit{asm\text{-}of}\ (\mathit{prog}_1)$, and $\mathit{asm\text{-}of}(\mathit{prog}) = \emptyset$ otherwise.

5.2 Process-local Security Condition

We use the security lattice $(\{\mathbb{L},\mathbb{H}\},\sqsubseteq,\sqcup,\sqcap)$ with $\mathbb{L} \sqsubseteq \mathbb{H}$ to express security policies for the variables local to an individual process. For a variable, the security level $\mathbb{H}$ (high) indicates that it may contain (or depend on) secrets, and the security level $\mathbb{L}$ indicates that it must not. We use the environment $\mathit{lev} : \mathit{Var} \to \mathit{Lev}$ to record the security level of each variable.

As usual, we define the *low-equivalence relation* $=_\mathbb{L}$ to capture agreement of memory states on low variables as follows.

Definition 2. $\mathit{mem}_1 =_\mathbb{L} \mathit{mem}_2 \triangleq \forall x \in \mathit{Var} : \mathit{lev}(x) = \mathbb{L} \Rightarrow \mathit{mem}_1(x) = \mathit{mem}_2(x)$

Hence, low-equivalent memory states can differ only in high variables.

For each $\mathit{acs} \in 2^{AsmCh}$, we define a relation $\overset{acs}{=\!=} \subseteq \Sigma \times \Sigma$ using the relations $\overset{ch}{=}_\#$ and $\overset{ch}{=}_1$, which are defined in Sect. 3.

Definition 3. $\sigma_1 \overset{acs}{=\!=} \sigma_2 \triangleq \forall \mathit{asm} \in \mathit{Asm} : \forall \mathit{ch} \in \mathit{ICh} : (\mathit{asm}, \mathit{ch}) \in \mathit{acs} \Rightarrow (\mathit{asm} = \mathbb{L}^\circ \Rightarrow \sigma_1 \overset{ch}{=}_\# \sigma_2) \wedge (\mathit{asm} = \mathbb{L}^\bullet \Rightarrow \sigma_1 \overset{ch}{=}_1 \sigma_2)$

The relation $\overset{acs}{=\!=}$ captures a similarity of channel states. If $\sigma_1 \overset{acs}{=\!=} \sigma_2$ and $(\mathbb{L}^\circ, \mathit{ch}) \in \mathit{acs}$, then the two channel states agree wrt. the availability of messages on the channel ch. If $\sigma_1 \overset{acs}{=\!=} \sigma_2$, $(\mathbb{L}^\bullet, \mathit{ch}) \in \mathit{acs}$, and both $\sigma_1(\mathit{ch})$ and $\sigma_2(\mathit{ch})$ are non-empty, then the first message available on ch is the same in σ_1 and σ_2. Note that Definition 3 imposes these restrictions only for internal channels.

Recall from Sect. 4 that the relation $\simeq_\Sigma$ captures an agreement of channel states wrt. the attacker's observations on external channels. We use $\simeq_\Sigma$ in combination with $\overset{acs}{=\!=}$ in our definition of local security.

We say that a channel state σ satisfies an assumption $(\mathit{NE}, \mathit{ch})$ if $|\sigma(\mathit{ch})| > 0$. We introduce a relation $\ltimes^{NE} \subseteq \Sigma \times 2^{AsmCh}$ to capture that all assumptions of the form $(\mathit{NE}, \mathit{ch})$ in a set of assumptions are satisfied.

Definition 4. $\sigma \ltimes^{NE} \mathit{acs} \triangleq \forall \mathit{ch} \in \mathit{Ch} : (\mathit{NE}, \mathit{ch}) \in \mathit{acs} \Rightarrow |\sigma(\mathit{ch})| > 0$

Our notion of local security builds on a notion of assumption-aware bisimilarity, that characterizes process configurations with similar observable data and behavior, despite potential differences in secrets.

Definition 5. *A symmetric relation R on process configurations is an* assumption-aware bisimulation*, if $\mathit{pcnf}_1\ R\ \mathit{pcnf}_2$, $\mathit{pcnf}_1 = \langle \mathit{prog}_1; \mathit{mem}_1 \rangle$, $\mathit{pcnf}_2 = \langle \mathit{prog}_2; \mathit{mem}_2 \rangle$, $\mathit{acs}_1 = \mathit{asm\text{-}of}(\mathit{prog}_1)$, and $\mathit{acs}_2 = \mathit{asm\text{-}of}(\mathit{prog}_2)$, imply $\mathit{mem}_1 =_\mathbb{L} \mathit{mem}_2$, $\mathit{prog}_1 = \mathsf{stop} \Leftrightarrow \mathit{prog}_2 = \mathsf{stop}$, and*

$$\begin{pmatrix} \langle \mathit{pcnf}_1; \sigma_1 \rangle \to \langle \mathit{pcnf}_1'; \sigma_1' \rangle \wedge \\ \sigma_1 \simeq_\Sigma \sigma_2 \wedge \sigma_1 \overset{\mathit{acs}_1 \cup \mathit{acs}_2}{=\!=\!=} \sigma_2 \wedge \\ \sigma_1 \ltimes^{NE} \mathit{acs}_1 \wedge \sigma_2 \ltimes^{NE} \mathit{acs}_2 \end{pmatrix} \Rightarrow \begin{pmatrix} \exists \mathit{pcnf}_2', \sigma_2' : \\ \langle \mathit{pcnf}_2; \sigma_2 \rangle \to \langle \mathit{pcnf}_2'; \sigma_2' \rangle \wedge \\ \sigma_1' \simeq_\Sigma \sigma_2' \wedge \mathit{pcnf}_1'\ R\ \mathit{pcnf}_2' \end{pmatrix}$$

We call two process configurations assumption-aware bisimilar *if there is an assumption-aware bisimulation in which they are related.*

If two process configurations are assumption-aware bisimilar, then differences between their memories can only exist in high variables. In addition, if one process is unable to take a step, then the other process is not able to either. Furthermore, after each lockstep taken by the respective processes under assumptions, differences in secrets over the channels and in the variables are not propagated to the low variables or to the observations that can be made by the attacker. It follows that no observable differences can result from differences in secrets after an arbitrary number of locksteps taken under assumptions. Note that channel states in the beginning of each lockstep are limited to ones that differ only in the channels over which $\mathbb{L}^\circ$ and $\mathbb{L}^\bullet$ are not used, and ones where the availability of messages agrees with the use of NE. By limiting the potential channel states, the potential environments of the program is more precisely characterized. Also note that the preservation of $\sigma_1 \overset{acs_1 \cup acs_2}{=\!=\!=} \sigma_2$ by each lockstep is not required. We explain why compositional reasoning using local security is sound in Sect. 5.5.

We define the relation $\approx$ to be the union of all assumption-aware bisimulations. It is easy to see that $\approx$ is again an assumption-aware bisimulation. Based on $\approx$, we define an indistinguishability relation on programs, and our notion of local security by classifying a program as secure if it is indistinguishable to itself.

Definition 6. *We say two programs $prog_1$ and $prog_2$ are* indistinguishable*, denoted by $prog_1 \sim prog_2$, if $mem_1 =_{\mathbb{L}} mem_2 \Rightarrow \langle prog_1; mem_1\rangle \approx \langle prog_2; mem_2\rangle$.*

Definition 7 (Local Security). *A program prog is* locally secure*, denoted by $LSec(prog)$, if $prog \sim prog$.*

Local security allows one to reason about a program in a restricted class of environments meeting explicit assumptions. This fine-grained treatment of environments provides formal underpinnings for the intuitive security of programs (e.g., ones in Sect. 2 and Sect. 6.3) that are deemed leaky by existing security properties (e.g., [5, 24, 25]) for message-passing communication or interaction.

Remark 2. Note that the local security of a program is preserved when a private external channel is turned into an internal channel, or vice versa, in case no assumption out of $\mathbb{L}^\bullet$ and $\mathbb{L}^\circ$ is made over the channel.

Remark 3. Note also that the local security of a program is not affected by using the assumptions $\mathbb{L}^\bullet$ and $\mathbb{L}^\circ$ over external channels. Technically, this is because of Definition 5, 6, and 7. That is, our definitions reflect a conservative approach to reasoning about open systems. In fact, we do not expect $\mathbb{L}^\bullet$ and $\mathbb{L}^\circ$ to be used over external channels.

5.3 Compositional Reasoning about Information-Flow Security

Our notion of local security allows one to decompose the verification of knowledge-based security for distributed programs into the verification of processes.

Theorem 1 (Soundness of Compositional Reasoning). *For a distributed program $dprog = ||_i prog_i$, if $LSec(prog_i)$ holds for all i, and $dprog$ ensures a sound use of assumptions, then we have $KBSec(dprog)$.*

This theorem requires that a distributed program ensures a sound use of assumptions. Intuitively, "sound use of assumptions" means that assumptions are met by the actual environment of the component program inside the distributed program. We will precisely define this notion in Sect. 5.4.

Since local security needs to be verified for less complex systems (i.e., processes), Theorem 1 reduces the conceptual complexity of verifying the security of distributed programs considerably. Since assumptions can be exploited when verifying local security, compositional reasoning based on Theorem 1 is also less restrictive than in existing work on systems with message passing (e.g., [7,8,25]).

5.4 Instrumented Semantics and Sound Use of Assumptions

We define the notion that a distributed program ensures a sound use of assumptions through an instrumentation of the semantics of our programming language.

We use *instrumentation states* to over-approximate the dependency of message content and presence on secrets over internal channels. Instrumentation states are from the set $InSt = \{\mu \in ICh \to (Lev^* \times Lev) \mid \mu(ch) = (llst, \mathbb{H}) \Rightarrow \exists n : llst = \mathbb{H}^n\}$. An instrumentation state maps each internal channel to a pair $(llst, \ell)$. Here $llst$ is a list of security levels, each for a message over the channel, while ℓ is the security level for the overall presence of messages over the channel. The level $\mathbb{H}$ (confidential) indicates that message content or presence might depend on secrets, while the level $\mathbb{L}$ (public) indicates that message content or presence must not depend on secrets. In case $\ell = \mathbb{H}$, all levels in $llst$ are $\mathbb{H}$, which reflects that presence cannot be more confidential than content (e.g., [24, 25]).

We use the relation $\models$ between $\Sigma \times InSt$ and $AsmCh$ to say that an assumption for a channel holds in a channel state and an instrumentation state.

Definition 8. *The relation* $\models$ *is defined by*

$$
\begin{aligned}
(\sigma, \mu) &\models (NE, ch) \text{ iff } |\sigma(ch)| > 0 \\
(\sigma, \mu) &\models (\mathbb{L}^\bullet, ch) \text{ iff } ch \in ICh \wedge \mu(ch) = (\ell \cdot llst, \ell') \Rightarrow \ell = \mathbb{L} \\
(\sigma, \mu) &\models (\mathbb{L}^\circ, ch) \text{ iff } ch \in ICh \wedge \mu(ch) = (llst, \ell') \Rightarrow \ell' = \mathbb{L}
\end{aligned}
$$

That is, the assumption NE on a channel ch holds if some message is available over ch. The assumption $\mathbb{L}^\bullet$ on an internal channel ch holds if the next message over ch is public. The assumption $\mathbb{L}^\circ$ on an internal channel ch holds if the presence of messages over ch is public. We lift $\models$ to sets of assumptions by $(\sigma, \mu) \models acs$ iff $\forall ac \in acs : (\sigma, \mu) \models ac$.

We use *rich traces* to explicate the update of instrumentation states in an execution of a distributed program. A rich trace rtr is a sequence of configurations of the form $\langle pcnfl; \sigma; \mu \rangle$, extending a global configuration $\langle pcnfl; \sigma \rangle$ with the instrumentation state μ. We use the expression $rtraces_\xi(dprog)$ to represent the set of possible rich traces of $dprog$ under the strategy ξ. We define $rtraces_\xi(dprog)$ based on an *instrumented semantics* (omitted here to save space), analogously to defining $traces_\xi(dprog)$ based on the basic semantics of our language in Sect. 3.

With the help of instrumentation states and rich traces, we precisely define "sound use of assumptions" as appearing in Theorem 1.

Definition 9. *The distributed program dprog* ensures a sound use of assumptions, *if* $rtr \cdot \langle[\langle prog_1; mem_1\rangle, \ldots, \langle prog_n; mem_n\rangle]; \sigma; \mu\rangle \in rtraces_\xi(dprog)$ *implies* $\forall j \in \{1, \ldots, n\} : (\sigma, \mu) \models asm\text{-}of(prog_j)$.

This definition characterizes the sound use of assumptions as a property over individual rich traces – as a safety property [15] rather than a hyperproperty [6].

Remark 4. How to verify the sound use of assumptions for information-flow security has been shown in [18]. The adaptation of the techniques in [18] to verify the sound use of *NE* and $\mathbb{L}^\circ$ is straightforward. For verifying the sound use of $\mathbb{L}^\bullet$, communication topology analyses [20] that approximate the correspondences between inputs and outputs can be built upon. If with each $\mathbb{L}^\bullet$ at a receive, the corresponding send comes with a public expression, then the use of $\mathbb{L}^\bullet$ is sound.

5.5 Soundness of Compositional Reasoning

We sketch our proof of Theorem 1. To begin with, we define a notion of low projection, on messages and message queues. A low projection reveals message content and presence without actual dependency on secrets according to the security levels tracked in the instrumentation states.

Definition 10. *We define the function* $\lfloor\cdot\rfloor_\ell : Val \to Val \cup \{\bigodot\}$ *for the low projection of a message under* $\ell \in Lev$, *by* $\lfloor v\rfloor_{\mathbb{L}} = v$ *and* $\lfloor v\rfloor_{\mathbb{H}} = \bigodot$.

The low projection of a message reveals the content or the mere existence ($\bigodot$) of the message depending on whether the content is public or confidential.

Definition 11. *We define the partial function* $\lfloor\cdot\rfloor_{(llst,\ell)} : Val^* \rightharpoonup (Val \cup \{\bigodot\})^* \cup \{\circledcirc\}$ *for the low projection of a message queue under* $llst \in Lev^*$ *and* $\ell \in Lev$, *by* $\lfloor\epsilon\rfloor_{(\epsilon,\mathbb{L})} = \epsilon$, $\lfloor vlst \cdot v\rfloor_{(llst\cdot\ell,\mathbb{L})} = \lfloor vlst\rfloor_{(llst,\mathbb{L})} \cdot \lfloor v\rfloor_\ell$, *and* $\lfloor vlst\rfloor_{(llst,\mathbb{H})} = \circledcirc$.

The low projection of a message queue reveals the content or existence of the individual messages, or the mere existence of the queue ($\circledcirc$), depending on whether the overall message presence is public or confidential.

The following result bridges equality of low projections of message queues in given channel and instrumentation states, and low-equivalence of the channel states under assumptions that hold in the channel and instrumentation states.

Proposition 1. *If* $\forall ch \in ICh : \lfloor\sigma_1(ch)\rfloor_{\mu_1(ch)} = \lfloor\sigma_2(ch)\rfloor_{\mu_2(ch)}$, $(\sigma_1, \mu_1) \models acs_1$, *and* $(\sigma_2, \mu_2) \models acs_2$, *then* $\sigma_1 \overset{acs_1 \cup acs_2}{=\!=\!=} \sigma_2$.

The essence of this proposition is: If message content or presence does not actually depend on secrets over an internal channel, then it is safe to reason under the assumption that message content or presence is public.

For a given distributed program $dprog = ||_i prog_i$, if we could show that for an arbitrary strategy ξ giving rise to a trace tr, each strategy ξ' indistinguishable to ξ gives rise to a trace tr' with the same observation to that of tr, then $KBSec(dprog)$ would follow immediately.

$$\frac{NE \notin as \Rightarrow lev^{\circ}(ch, as) = \mathbb{L} \quad lev^{\circ}(ch, as) \sqcup lev^{\bullet}(ch, as) \sqsubseteq lev(x)}{lev \vdash {}^{as}\mathsf{recv}(ch, x)} \qquad \frac{NE \notin as \Rightarrow lev^{\circ}(ch, as) \sqsubseteq lev(x_{\mathrm{b}}) \quad lev^{\circ}(ch, as) \sqcup lev^{\bullet}(ch, as) \sqsubseteq lev(x)}{lev \vdash {}^{as}\mathsf{if\text{-}recv}(ch, x, x_{\mathrm{b}})}$$

$$\frac{lev\langle e \rangle \sqsubseteq lev^{\bullet}(ch, \emptyset)}{lev \vdash \mathsf{send}(ch, e)} \qquad \frac{lev\langle e \rangle \sqsubseteq lev(x)}{lev \vdash x := e} \qquad \frac{}{lev \vdash \mathsf{skip}} \qquad \frac{lev \vdash prog_1 \quad lev \vdash prog_2}{lev \vdash prog_1; prog_2}$$

$$\frac{lev \vdash prog_1 \quad lev \vdash prog_2 \quad lev\langle e \rangle = \mathbb{H} \Rightarrow prog_1 \sim prog_2}{lev \vdash \mathsf{if}\ e\ \mathsf{then}\ prog_1\ \mathsf{else}\ prog_2\ \mathsf{fi}} \qquad \frac{lev\langle e \rangle = \mathbb{L} \quad lev \vdash prog}{lev \vdash \mathsf{while}\ e\ \mathsf{do}\ prog\ \mathsf{od}}$$

$$lev^{\bullet}(ch, as) = \begin{cases} \mathbb{L} & \text{if } ch \in ICh \wedge \mathbb{L}^{\bullet} \in as \\ & \quad \vee ch \in PubCh \\ \mathbb{H} & \text{otherwise} \end{cases} \qquad lev^{\circ}(ch, as) = \begin{cases} \mathbb{L} & \text{if } ch \in ICh \wedge \mathbb{L}^{\circ} \in as \vee \\ & \quad ch \in PubCh \vee ch \in EncCh \\ \mathbb{H} & \text{otherwise} \end{cases}$$

Fig. 1: The Security Type System

Since $LSec(prog_i)$ holds for each program $prog_i$ by the hypotheses of Theorem 1, and $mem_{\mathrm{init}} =_{\mathbb{L}} mem_{\mathrm{init}}$ holds, there exists an assumption-aware bisimulation relating each process configuration $\langle prog_i; mem_{\mathrm{init}} \rangle$ to itself. We show the existence of tr' by simulating each step in tr taken by the program $prog_i$ in the assumption-aware bisimulation for $prog_i$. As a key part of establishing this single-step simulation, we establish the condition $\sigma_1 \overset{acs_1 \cup acs_2}{=\!=\!=} \sigma_2$ in the corresponding assumption-aware bisimulation using Proposition 1. Among the hypotheses of this proposition, we obtain $(\sigma_1, \mu_1) \models acs_1$ and $(\sigma_2, \mu_2) \models acs_2$ by the hypothesis of Theorem 1 that *dprog* ensures a sound use of assumptions. We establish $\forall ch \in ICh : \lfloor \sigma_1(ch) \rfloor_{\mu_1(ch)} = \lfloor \sigma_2(ch) \rfloor_{\mu_2(ch)}$ as a property enforced by our instrumented semantics. For space reasons, we omit the details of the proof.

6 Security Type System and Evaluation

In information-flow security, it is often not straightforward to verify a system by directly applying the security definition. In our case, it is not straightforward to verify the local security of a program by constructing an assumption-aware bisimulation (although this is possible). Security type systems [28] are an effective technique for the verification of noninterference properties [26]. Local security (cf. Sect. 5) not only permits compositional reasoning about global security (cf. Sect. 4), but also admits sound verification using security type systems. We devise a sound security type system for local security, and evaluate our type system using more practical examples than in Sect. 2.

6.1 Judgment and Typing Rules

Our security type system establishes the judgment $lev \vdash prog$, which says that the program *prog* is well-typed given the environment $lev : Var \to Lev$.

The typing rules are presented in Fig. 1. They are formulated with the auxiliary functions $lev^{\bullet}$ and lev° of type $(Ch \times 2^{Asm}) \to Lev$. The function $lev^{\bullet}$ gives

the content levels of channels according to the protection of external channels, and to the potential assumption $\mathbb{L}^\bullet$ for internal channels. The function lev° gives the presence levels of channels according to the protection of external channels, and to the potential assumption $\mathbb{L}^\circ$ for internal channels.

The first premise of the rule for ${}^{as}\mathsf{recv}(ch, x)$ says that without the assumption NE, message presence over ch is required to be public. The first premise of the rule for ${}^{as}\mathsf{if\text{-}recv}(ch, x, x_\mathrm{b})$ says that without NE, message presence over ch is required to be no more confidential than the flag variable x_b. The intuition for these two premises is: If the receiving channel is non-empty, the blocking receive cannot block, and the non-blocking receive always completes with x_b set to true; hence no information leakage occurs via blocking or via x_b. The second premise in each of the receive rules says that the receiving variable x is more confidential than the presence and content of messages over the receiving channel ch. This premise reflects that the value of x after the receive could reflect both the presence and the content of messages over ch at the receive. The premise allows relaxing the typing constraint on x with the use of $\mathbb{L}^\bullet$ and $\mathbb{L}^\circ$ that signals temporarily public message content and presence over an internal receiving channel. For $\mathsf{send}(ch, e)$, the only premise is concerned with message content. This is because a send never blocks, and hence does not leak information via blockage. For a conditional, in case the guard is confidential, the branches are required to be indistinguishable, since which branch is executed might be affected by secrets. Note that the indistinguishability of the branches is a semantic condition. In an implementation of the rule for conditionals, one would not directly check this indistinguishability, but would rather check a syntactic approximation of it. In an addendum of this paper, we present a type system for local security, where syntactic approximation is employed via the notion of low slices (e.g., [18]).[1] For a loop, the typing rule requires the looping condition to be public, for ensuring progress-sensitive security. The remaining rules in Fig. 1 are straightforward.

Our security type system is novel in exploiting assumptions to enable permissive type-checking. The assumption NE makes type-checking permissive by providing the information that a receive cannot block. In addition, the assumptions $\mathbb{L}^\bullet$ and $\mathbb{L}^\circ$ make type-checking permissive by enabling content-sensitivity (varying content confidentiality over one single channel) and availability-sensitivity (varying presence confidentiality over one single channel).

Our security type system is syntax-directed, i.e., the choice of an applicable typing rule is deterministic for each given program. Hence, the implementation of our security type system (with the syntactic approximation) is straightforward.

Remark 5. Note that the type system in Fig. 1 is not flow-sensitive [12]. Flow-sensitivity could be added by allowing separate typing environments for variables at different program points as in [12]. This extension would, however, unnecessarily complicate our technical exposition.

[1] The addendum can be found at `http://www.mais.informatik.tu-darmstadt.de/WebBibPHP/papers/2017/LiMantelTasch-addendum-APLAS2017.pdf`

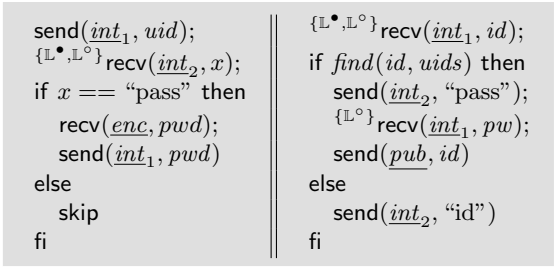

Fig. 2: Authentication System $auth = [auth\text{-}cl \parallel auth\text{-}srv]$

6.2 Soundness

Our type system is sound: The typability of a given program implies that the program is secure in the sense of Definition 7.

Theorem 2 (Soundness). *If $lev \vdash prog$, then $LSec(prog)$.*

This theorem permits sound reasoning about local security at a syntactic level, exploiting assumptions at receive commands as described by our type system.

The following corollary says that the security of a distributed program can be verified by locally typing its constituent programs under suitable assumptions, and globally verifying the sound use of these assumptions. This corollary immediately follows from Theorem 1 and Theorem 2.

Corollary 1. *For a distributed program $dprog = ||_i prog_i$, if $lev \vdash prog_i$ for all i, and $dprog$ ensures a sound use of assumptions, then $KBSec(dprog)$.*

This corollary replaces the semantic condition of local security in Theorem 1 with the syntactic condition of typability. It provides a path to automated security analysis for distributed programs that enjoys both modularity and precision.

6.3 Examples of Typable Programs

All example programs in Sect. 2 are typable with suitable use of assumptions. We further evaluate our security type system using two additional examples that more closely reflect communication patterns identifiable in real-world applications. In both examples, we preclude the possibility for the attacker to directly obtain information via wiretapping. Correspondingly, we model some of the communication lines as internal channels represented by $\underline{int}_1$ and $\underline{int}_2$.

Example: Authentication. Consider the distributed program *auth* in Fig. 2. It captures an authentication scenario where the server process executes the program on the right, and the client process executes the program on the left.

The server receives the user's ID over the channel $\underline{int}_1$. If the ID exists in the database, it requests the user's password. If it then receives a password, it forwards the ID over the channel $\underline{pub}$ to a logging process that maintains a log in the environment. The log needs to be analyzable by parties not entitled to know the passwords, for e.g., detecting and tracing potential brute-force attacks.

The client sends the user's ID over the channel $\underline{int}_1$ to the server. If it then receives a password request, the client receives the user's secret password over the channel $\underline{enc}$, and forwards it over $\underline{int}_1$. Otherwise, it terminates.

Annotation. The authentication server receives both the public user ID and the secret password over the channel $\underline{int}_1$. The assumption $\mathbb{L}^\bullet$ at $\mathsf{recv}(\underline{int}_1, id)$ in the server program captures that the user ID provided by the environment (the client) is expected to be public. On the other hand, the default invisibility of $\underline{int}_1$ is in line with the password to be received at $\mathsf{recv}(\underline{int}_1, pw)$ being confidential.

Typability. Supposing $lev(id) = lev(uids) = lev(uid) = lev(x) = \mathbb{L}$ and $lev(pw) = lev(pwd) = \mathbb{H}$, we explain how the server program *auth-srv* is type checked. For ${}^{\{\mathbb{L}^\bullet,\mathbb{L}^\circ\}}\mathsf{recv}(\underline{int}_1, id)$, since $NE \notin \{\mathbb{L}^\bullet, \mathbb{L}^\circ\}$, the first premise of the receive rule requires $lev^\circ(\underline{int}_1, \{\mathbb{L}^\bullet, \mathbb{L}^\circ\}) = \mathbb{L}$, which holds. The second premise of the receive rule is also satisfied since $lev^\circ(\underline{int}_1, \{\mathbb{L}^\bullet, \mathbb{L}^\circ\}) \sqcup lev^\bullet(\underline{int}_1, \{\mathbb{L}^\bullet, \mathbb{L}^\circ\}) = \mathbb{L}$, which allows id to be public. For ${}^{\{\mathbb{L}^\circ\}}\mathsf{recv}(\underline{int}_1, pw)$, since $lev^\circ(\underline{int}_1, \{\mathbb{L}^\circ\}) = \mathbb{L}$, the first premise of the receive rule is satisfied. Since $lev(pw) = \mathbb{H}$, the second premise is also satisfied. Without going into further details, we claim that both *auth-srv* and *auth-cl* are typable.

Sound Use of Assumptions. Using our instrumented semantics, we can verify that *auth* ensures a sound use of assumptions. Intuitively, the use of $\mathbb{L}^\circ$ is sound because neither *auth-cl* nor *auth-srv* contains high branches or loops that might cause sending to an internal channel in a secret-dependent fashion. The use of $\mathbb{L}^\bullet$ is sound because the only secret message is sent at $\mathsf{send}(\underline{int}_1, pwd)$. This message cannot be received at any receive command annotated with $\mathbb{L}^\bullet$.

Knowledge-based Security. Corollary 1 allows us to conclude $KBSec(auth)$.

Remark 6 (An ill-typed variant). Note that if the assumption $\mathbb{L}^\bullet$ in the receive command ${}^{\{\mathbb{L}^\bullet,\mathbb{L}^\circ\}}\mathsf{recv}(\underline{int}_1, id)$ had been missing, this command would have been ill-typed. In this situation, the message to be received over $\underline{int}_1$ is expected to be confidential, which conflicts with the receiving variable id being public.

Example: Auction. Consider the distributed program *auct* in Fig. 3. It captures a single-bid sealed auction where a bidder is allowed to blindly submit a sole bid for an auction. The server process handling the auction executes the program on the right, and the client process executes the program on the left.

The server waits for the bidder's registration over $\underline{int}_1$ (for simplicity represented abstractly by "reg"). Afterwards, it forwards the registration over $\underline{pub}$ to a publicly accessible bulletin board. Then it sends the minimal bid (i.e., the price below which the item cannot be sold) to the bidder over $\underline{int}_2$. If it receives a bid over $\underline{int}_1$, it forwards this bid over $\underline{pri}$ to the process determining the outcome of the auction. Otherwise, it terminates.

The client's behavior depends on a secret threshold for placing a bid. The client receives this threshold *thres* over $\underline{enc}$. Afterwards, it registers for the auction over $\underline{int}_1$. If the minimal bid received over $\underline{int}_2$ does not exceed the threshold *thres*, the client computes and sends a bid over $\underline{int}_1$. Otherwise, it terminates.

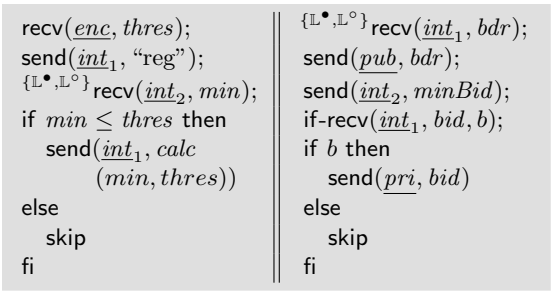

Fig. 3: Auction System $auct = [auct\text{-}cl \parallel auct\text{-}srv]$

Annotation. Over the channel $\underline{int}_1$, the auction server receives both a registration whose presence is public, and a bid whose presence is confidential (to keep the threshold a secret). The assumption $\mathbb{L}^\circ$ at $^{\{\mathbb{L}^\bullet,\mathbb{L}^\circ\}}\mathsf{recv}(\underline{int}_1, bdr)$ in the server program captures that the presence of a registration coming from the environment (the client) is expected to be public. On the other hand, the default invisibility of $\underline{int}_1$ is in line with the presence of a bid being confidential.

Typability. Supposing $lev(bdr) = lev(minBid) = lev(min) = \mathbb{L}$ and $lev(bid) = lev(b) = lev(thres) = \mathbb{H}$, we explain how the server program *auct-srv* is type checked. The typing derivation of $^{\{\mathbb{L}^\bullet,\mathbb{L}^\circ\}}\mathsf{recv}(\underline{int}_1, bdr)$ is analogous to the typing derivation of $^{\{\mathbb{L}^\bullet,\mathbb{L}^\circ\}}\mathsf{recv}(\underline{int}_1, id)$ in the authentication example. For the command $\mathsf{if\text{-}recv}(\underline{int}_1, bid, b)$, since $NE \notin \emptyset$, the first premise of the rule for non-blocking receive requires $lev^\circ(\underline{int}_1, \emptyset) \sqsubseteq lev(b)$. This constraint is satisfied since $lev(b) = \mathbb{H}$. The second premise is also satisfied since $lev(bid) = \mathbb{H}$. Because $lev(b) = \mathbb{H}$, $\mathsf{send}(\underline{pri}, bid)$ and skip are required to be indistinguishable by the rule for conditional branching. Their indistinguishability can be established by constructing a suitable assumption-aware bisimulation. Without going into further details, we claim that both *auct-srv* and *auct-cl* are typable.

Sound Use of Assumptions. Using our instrumented semantics, we can verify that *auct* ensures a sound use of assumptions. Below, we provide the intuition for the sound use of $\mathbb{L}^\circ$. The only possibility for message presence to become dependent on secrets over an internal channel is via the send command over $\underline{int}_1$ in the high branching on $min \leq thres$. However, the only receive over $\underline{int}_1$ with the assumption $\mathbb{L}^\circ$ is always executed before this high branching is entered.

Knowledge-based Security. Corollary 1 allows us to conclude $KBSec(auct)$.

Remark 7 (An ill-typed variant). Note that if the assumption $\mathbb{L}^\circ$ in the receive command $^{\{\mathbb{L}^\bullet,\mathbb{L}^\circ\}}\mathsf{recv}(\underline{int}_1, bdr)$ had been missing, this command would have been ill-typed. In this situation, the presence of a message over $\underline{int}_1$ is expected to be confidential, which conflicts with the receiving variable *bdr* being public.

7 Related Work

Rely-Guarantee-style Reasoning for Information-Flow Security. The first development on rely-guarantee-style reasoning for information-flow security is [19]. It permits exploiting *no-read* and *no-write* assumptions on shared variables to achieve modular yet permissive security verification for shared-memory concurrent programs. The verification of the sound use of assumptions is addressed using Dynamic Pushdown Networks in [18].

The exploitation of assumptions for shared-memory concurrent programs has been developed further. In [3], a hybrid security monitor is proposed, while in [22], a static verification of value-sensitive security policies is proposed.

In [10], service-based systems with unidirectional invocation of services are addressed. While components communicate via calls and responses of services, the services of a component communicate via shared memory. Rely-guarantee-style reasoning is employed for the inter-component invocation of services.

Securing Communication in Distributed Systems. There are several approaches to statically securing communication in distributed systems. In [25], modular reasoning without exploiting assumptions is proposed. In [7], explicit treatment of cryptography is introduced and a property similar in spirit to that of [25] is enforced. In [1], "fast simulations" are proposed to guarantee termination-insensitive security under observational determinism [29], and the flow constraints needed to secure a system are thereby relaxed. No modular reasoning is supported in [1]. The closest development to rely-guarantee-style reasoning about message-passing communication (as in this article) might be [10]. In [10], the communication pattern is highly restricted: Communication is only used to model argument-passing and result-retrieval for remote procedure calls.

Security Type Systems. Security type systems (see, e.g., [26]) are a prominent class of techniques for the static verification of information-flow security. We only discuss closely related developments in the literature.

For message content, value-sensitive flow policies (e.g., [2,16]) and first class information-flow labels (e.g., [30]) in principle enable the treatment of channels with varying content confidentiality. Value-sensitive policies require the code to contain variables that regulate the security policies. First-class labels require built-in language support. Moreover, the actual transmission of the labels could lead to increased attack surface as well as performance overhead.

For message presence, the type systems of [5, 11, 14] enforce security properties concerned with the *permanent* blockage of input caused by secrets. To relax typing constraints wrt. message presence, linear types are used in [11], and a deadlock detection mechanism is used in [14]. In comparison, we enforce a stronger security property sensitive to the extent that public communication can be delayed, and relax typing constraints wrt. message presence by exploiting assumptions. The type system of [25] rejects all blocking inputs over high presence channels. A coarse-grained way of adjusting presence levels to make the analysis succeed is implicitly provided. The type system of [24] allows high presence input to be followed by low assignment, but not low output. Thus, the relaxation

of presence constraints in our type system and that of [24] is incomparable. No additional precision via availability-sensitivity is provided in [24, 25].

8 Conclusion

Our aim has been to resolve the imprecision problem in the modular reasoning about information-flow security for distributed systems, in the presence of message-passing communication. Our solution consists of a process-local security condition that soundly modularizes global security for distributed programs, and a security type system that soundly enforces the process-local condition. By exploiting assumptions about the interface of each process, our solution enables the relaxation of information-flow constraints concerned with the blockage of communication, and allows the treatment of channels that are content-sensitive and availability-sensitive at both the semantic level and the syntactic level.

Our development is performed for a simplified language. The adaption of our approach to real-world languages such as C and Java requires the treatment of features such as procedure calls and heaps. These features have been considered in existing type-based analyses of information-flow security (e.g., [23]), and we expect the treatment to be orthogonal to our treatment of communication. Furthermore, our use of assumptions is non-intrusive, the annotation of assumptions does not introduce changes to the underlying programming language. This non-intrusiveness facilitates the adaption of our approach to a realistic language.

A potential direction for future work is type inference. For our security type system, the inference of security levels for variables is straightforward – it can be performed in the same manner as for sequential languages. On the other hand, the inference of assumption annotations on receive commands is a separate research problem that we plan to address in the future.

Acknowledgments. The authors thank the anonymous reviewers for their helpful comments. This work was supported partially by the DFG under project RSCP (MA 3326/4-2/3) in the priority program RS3 (SPP 1496), and partially by the German Federal Ministry of Education and Research (BMBF) as well as by the Hessen State Ministry for Higher Education, Research and the Arts (HMWK) within CRISP.

References

1. R. Alpízar and G. Smith. Secure information flow for distributed systems. In *FAST'09*, pages 126–140.
2. T. Amtoft, J. Hatcliff, E. Rodríguez, Robby, J. Hoag, and D. Greve. Specification and checking of software contracts for conditional information flow. In *FM'08*.
3. A. Askarov, S. Chong, and H. Mantel. Hybrid monitors for concurrent noninterference. In *CSF'15*, pages 137–151.
4. A. Askarov and A. Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *S&P'07*, pages 207–221.

5. S. Capecchi, I. Castellani, M. Dezani-Ciancaglini, and T. Rezk. Session types for access and information flow control. In *CONCUR'10*, pages 237–252.
6. M. R. Clarkson and F. B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.
7. R. Focardi and M. Centenaro. Information flow security of multi-threaded distributed programs. In *PLAS'08*, pages 113–124.
8. R. Focardi and R. Gorrieri. Classification of security properties (part I: information flow). In *FOSAD'00*, pages 331–396.
9. J. A. Goguen and J. Meseguer. Security policies and security models. In *S&P'82*. IEEE Computer Society.
10. S. Greiner and D. Grahl. Non-interference with what-declassification in component-based systems. In *CSF'16*, pages 253–267.
11. K. Honda, V. T. Vasconcelos, and N. Yoshida. Secure information flow as typed process behaviour. In *ESOP'00*, pages 180–199.
12. S. Hunt and D. Sands. On flow-sensitive security types. In *POPL'06*, pages 79–90.
13. C. B. Jones. *Development methods for computer programs including a notion of interference*. Oxford University Computing Laboratory, 1981.
14. N. Kobayashi. Type-based information flow analysis for the pi-calculus. *Acta Inf.*, 42(4-5):291–347, 2005.
15. L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2):125–143, 1977.
16. X. Li, F. Nielson, H. R. Nielson, and X. Feng. Disjunctive information flow for communicating processes. In *TGC'15*, pages 95–111.
17. H. Mantel. Information flow and noninterference. In *Encyclopedia of Cryptography and Security, 2nd Ed.*, pages 605–607. 2011.
18. H. Mantel, M. Müller-Olm, M. Perner, and A. Wenner. Using dynamic pushdown networks to automate a modular information-flow analysis. In *LOPSTR'15*.
19. H. Mantel, D. Sands, and H. Sudbrock. Assumptions and guarantees for compositional noninterference. In *CSF'11*, pages 218–232.
20. M. Martel and M. Gengler. Communication topology analysis for concurrent programs. In *SPIN'00*, pages 265–286.
21. D. McCullough. A hookup theorem for multilevel security. *IEEE Trans. Software Eng.*, 16(6):563–568, 1990.
22. T. C. Murray, R. Sison, E. Pierzchalski, and C. Rizkallah. Compositional verification and refinement of concurrent value-dependent noninterference. In *CSF'16*.
23. A. C. Myers and B. Liskov. A decentralized model for information flow control. In *SOSP'97*, pages 129–142.
24. W. Rafnsson, D. Hedin, and A. Sabelfeld. Securing interactive programs. In *CSF'12*, pages 293–307.
25. A. Sabelfeld and H. Mantel. Securing communication in a concurrent language. In *SAS'02*, pages 376–394.
26. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.
27. A. S. Tanenbaum and M. van Steen. *Distributed systems - principles and paradigms, 2nd Edition*. 2007.
28. D. M. Volpano, C. E. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2/3):167–188, 1996.
29. S. Zdancewic and A. C. Myers. Observational determinism for concurrent program security. In *CSFW'03*, pages 29–43.
30. L. Zheng and A. C. Myers. Dynamic security labels and static information flow control. *Int. J. Inf. Sec.*, 6(2-3):67–84, 2007.