Secure Enforcement for Global Process Specifications

Jérémy Planul[1], Ricardo Corin[1], and Cédric Fournet[2,1]

[1] MSR-INRIA
[2] Microsoft Research

Abstract. Distributed applications may be specified as parallel compositions of processes that summarize their global interactions and hide local implementation details. These processes define a fixed protocol (also known as a contract, or a session) which may be used for instance to compile or verify code for these applications.
Security is a concern when the runtime environment for these applications is not fully trusted. Then, parts of their implementation may run on remote, corrupted machines, which do not comply with the global process specification. To mitigate this concern, one may write defensive implementations that monitor the application run and perform cryptographic checks. However, hand crafting such implementations is ad hoc and error-prone.
We develop a theory of secure implementations for process specifications. We propose a generic defensive implementation scheme, relying on history-tracking mechanisms, and we identify sufficient conditions on processes, expressed as a new type system, that ensure that our implementation is secure for all integrity properties. We illustrate our approach on a series of examples and special cases, including an existing implementation for sequential multiparty sessions.

1 Introduction

Distributed applications may be specified using concurrent processes that capture their global interactions, or *protocol*, and otherwise ignore their local implementation details. Hence, each machine that takes part in the application is assigned a *fixed initial process* (often called a *role* of the protocol) and may run any local code that implements its process, under the global assumption that all other participating machines will also comply with their respective assigned processes. This approach yields strong static guarantees, cuts the number of cases to consider at runtime, and thus simplifies distributed programming. It has been explored using (binary) sessions [6,5] and, more recently, *multiparty sessions* [3,7]. Within sessions, machines exchange messages according to fixed, pre-agreed patterns, for instance a sequence of inputs and outputs between a client and a server; these patterns may be captured by types, expressed as declarative contracts (also known as workflows), or more generally specified as processes.

Global process specifications also provide an adequate level of abstraction to address distributed security concerns. Each machine is then interpreted as a unit

To appear in the proceedings of the 20th International Conference on Concurrency Theory, M. Bravetti and G. Zavattato (Eds.): CONCUR 2009, LNCS

of trust, under the control of a principal authorized to run a particular role in the application. Conversely, these principals do not necessarily trust one another. Indeed, many applications commonly involve unknown parties on open networks, and a machine may participate in a session run that also involves untrustworthy machines (either malicious, or compromised, or poorly programmed). In this situation, we still expect *session integrity* to hold: an "honest" machine should accept a message as genuine only if it is enabled by the global specification. To this end, their implementation may perform cryptographic checks, using for example message signatures, as well as checks of causal dependencies between messages.

Cryptographic implementation mechanisms need not appear in the global specification. They can sometimes be automatically generated from this specification. Corin *et al.* perform an initial step in this direction, by securing implementations of n-ary sequential sessions [3,1]. Their sessions specify sequences of communications between n parties as paths in directed graphs. A global session graph is compiled down to secure local implementations for each role, using a custom cryptographic protocol that protects messages and monitors their dependencies. Their main security result is that session integrity is guaranteed for all session runs—even those that involve compromised participants. They also report on prototype implementations, showing that the protocol overhead is small. Although their sequential sessions are simple and intuitive, they also lack flexibility, and are sometimes too restrictive. For instance, concurrent specifications must be decomposed into series of smaller, sequential sessions, and the programmer is left to infer any security properties that span several sub-sessions. More generally, they leave the applicability of their approach to more expressive specification languages as an open problem.

In this paper, we consider a much larger class of session specifications that enable concurrency and synchronization within session runs; and we systematically explore their secure implementation mechanisms, thereby enforcing more global properties in a distributed setting. We aim at supporting arbitrary processes, and as such we depart significantly from prior work: our specification language is closer to a generic process algebraic setting, such as CCS [8]. On the other hand, we leave the cryptography implicit and do not attempt to generalize their concrete protocol design.

Example 1. To illustrate our approach, consider a simple model of an election, with three voting machines (V_1, V_2, and V_3) and one election officer machine (E) in charge of counting the votes for two possible candidates (c_1 and c_2) and sending the result (r_1 or r_2) to a receiver machine (R). We specify the possible actions of each machine using CCS-like processes (defined in Section 2), as follows:

$$V_1 = V_2 = V_3 = \overline{c_1} + \overline{c_2} \qquad E = c_1.c_1.\overline{r_1} \mid c_2.c_2.\overline{r_2} \qquad R = r_1 + r_2$$

Machines running process V_i for $i = 1, 2, 3$ may vote for one of the two candidates by emitting one of the two messages c_1 or c_2; this is expressed as a choice between two output actions. The machine running process E waits for two votes for the same candidate and then fires the result; this is expressed as a parallel

composition of two inputs followed by an output action. Finally, the machine running process R waits for one of the two outcomes; this is expressed as a choice between two input actions.

Protocols and applications We now give an overview of the protocols and the attacker model. Our processes represent protocol specifications, rather than the application code that would run on top of the protocol. Accordingly, our protocol implementation monitors and protects high-level actions driven by an abstract application. The implementation immediately delivers to the application any input action enabled by the specification, and sends to other machines any action output enabled by the specification and selected by the application. The application is in charge of resolving internal choices between different message outputs enabled by the specification, and to provide the message payloads.

We intend that our implementations meet two design constraints: (1) they rely only on the global specification—not on the knowledge of which machines have been compromised, or the mediation of a trusted third party; and (2) they do not introduce any extra message: each high-level communication is mapped to an implementation message. This *message transparency* constraint excludes unrealistic implementations, such as a global, synchronized implementation that reaches a distributed consensus on each action.

Attacker Model We are interested in the security of any subset of machines that run our implementation, under the assumption that the other machines may be corrupted. In Example 1, for instance, the receiver machine should never accept a result from the election officer if no voter has cast its vote.

Our implementations provide protection against active adversaries that controls parts of the session run: We assume an unsafe network, so the adversary may intercept, reorder, and inject messages—this is in line with classic symbolic models of cryptography pioneered by Dolev and Yao [4]. We also assume partial compromise, so the adversary may control some of the machines supposed to run our implementation, and run instead arbitrary code; hence, those machines need not follow their prescribed roles in the session specification and may instead collude to forge messages in an attempt to confuse the remaining compliant, honest machines.

We focus on global control flow integrity, rather than cryptographic wire formats; thus, we assume that the message components produced by the compliant machines cannot be forged by our adversary. (This can be achieved by standard cryptographic integrity mechanisms, such as digital signatures.) Conversely, the adversary controls the network and may forge any message component from compromised machines. Also, we do not address other security properties of interest, such as payload confidentiality or anonymity.

Contributions Relying on process calculi, we define an expressive language for specifications and formally describe its implementation. We construct a secure, generic implementation scheme: we propose a general implementability condition, expressed as a type system, and show that it suffices to ensure that any set of compliant machines remain in a globally-consistent state, despite any coordinated attack by an adversary in control of the remaining, corrupted machines.

Contents Section 2 defines a global semantics for specifications. Section 3 describes their generic implementations, and states their soundness and completeness properties. Section 4 considers binary specifications. Section 5 defines our type system. Section 6 presents our history-tracking implementation and establishes its correctness for well-typed specifications. Section 7 considers sequential n-ary sessions. Section 8 concludes. Additional details and proofs appear in an online paper at `http://msr-inria.inria.fr/projects/sec/sessions`.

2 Global process specifications

We consider specifications that consist of distributed parallel compositions of local processes, each process running on its own machine. In the remainder of the paper, we let n be a fixed number of machines, and let $\widetilde{P}$ range over global process specifications, that is, n-tuples of local processes $(P_0, \ldots, P_i, \ldots, P_{n-1})$.

Syntax and informal semantics Our local processes use a CCS syntax, given below. Their outputs are asynchronous, since we are in a distributed setting.

$P ::=$	Local processes
$\mathbf{0}$	inert process
$\overline{a}$	asynchronous send
$a.P$	asynchronous receive
$P + P'$	choice
$P \mid P'$	parallel fork
$!P$	replication

The specification $\widetilde{P} = (P_0, \ldots, P_{n-1})$ sets a global "contract" between n machines; it dictates that each machine i behaves as specified by P_i. For instance, in Example 1 we have $n = 5$ and $\widetilde{P} = (V_1, V_2, V_3, E, R)$.

Operational semantics ($\rightarrow_{\mathsf{P}}$ and $\rightarrow_{\mathsf{S}}$) We define standard labelled transitions for local processes, with the rules given below. We write $P \xrightarrow{\beta}_{\mathsf{P}} P'$ when process P evolves to P' with action β ranging over a and $\overline{a}$. We omit the symmetric rules for sum and parallel composition.

$$a.P \xrightarrow{a}_{\mathsf{P}} P \qquad \overline{a} \xrightarrow{\overline{a}}_{\mathsf{P}} 0 \qquad \frac{P_0 \xrightarrow{\beta}_{\mathsf{P}} P_0'}{P_0 + P_1 \xrightarrow{\beta}_{\mathsf{P}} P_0'} \qquad \frac{P_0 \xrightarrow{\beta}_{\mathsf{P}} P_0'}{P_0 | P_1 \xrightarrow{\beta}_{\mathsf{P}} P_0' | P_1} \qquad \frac{P \xrightarrow{\beta}_{\mathsf{P}} P'}{!P \xrightarrow{\beta}_{\mathsf{P}} !P | P'}$$

We also define labelled transitions for global configurations, with the communication rule below. We write $\widetilde{P} \xrightarrow{i\ a\ j}_{\mathsf{S}} \widetilde{P}'$ when $\widetilde{P}$ evolves to $\widetilde{P}'$ by action a with sender P_i and receiver P_j. (The case $i = j$ is for local, but still observable communications.) We let α range over global communication labels $i\ a\ j$, and let φ range over sequences of these labels (written for instance $i\ a\ j.\alpha$) representing high-level traces. We write $\alpha \in \varphi$ when α occurs in φ.

$$\frac{P_i \xrightarrow{\overline{a}}_{\mathsf{P}} P_i^\circ \quad (P_k = P_k^\circ)^{k \neq i} \quad P_j^\circ \xrightarrow{a}_{\mathsf{P}} P_j' \quad (P_k^\circ = P_k')^{k \neq j}}{\widetilde{P} \xrightarrow{i\ a\ j}_{\mathsf{S}} \widetilde{P}'}$$

3 Distributed process implementations

We describe distributed implementations of the specifications of Section 2, each process being mapped to one machine. We separate compliant (honest) machines from compromised (dishonest) machines, then we define their implementations, their semantics, and their properties (soundness and completeness).

We let $\mathcal{C}$ range over subsets of $\{0, \ldots, n-1\}$, representing the indexes of $\widetilde{P}$ whose machines are compliant. We let $\widehat{P}$ range over tuples of processes indexed by $\mathcal{C}$. Intuitively, these machines follow our implementation semantics, whereas the other machines are assumed to be compromised, and may jointly attempt to make the compliant machines deviate from the specification $\widetilde{P}$. For instance, if the election officer of Example 1 is compromised, it may immediately issue a result $\overline{r}_1$ without waiting for two votes c_1. Similarly, a compromised voter may attempt to vote twice. We are interested in protecting (the implementations of) compliant machines from such actions.

We give a generic definition of process implementations. In the following sections, we show how to instantiate this definition for a given specification. Informally, an implementation is a set of programs, one for each specification process to implement, plus a definition of the messages that the adversary may be able to construct, in the spirit of symbolic models for cryptography.

Definition 1 (Distributed implementation). *A* distributed implementation *is a triple* $\langle (Q_i)_{i<n}, (\xrightarrow{\gamma}_i)_{i<n}, (\vdash_{\mathcal{C}})_{\mathcal{C}\subseteq 0..n-1} \rangle$ *(abbreviated* $\langle \widetilde{Q}, \widetilde{\rightarrow}, \vdash_{\mathcal{C}} \rangle$*) where, for each* $i \in 0..n-1$ *and* $\mathcal{C} \subseteq 0..n-1$*,*

- Q_i *is an implementation process;*
- $\xrightarrow{\gamma}_i$ *is a labelled transition relation between implementation processes;*
- $\vdash_{\mathcal{C}}$ *is a relation between message sets and messages.*

In the definition, each Q_i is an initial implementation process, and each $\xrightarrow{\gamma}_i$ is a specialized transition relation between implementation processes for machine i, labelled with either an input (M) or an output ($\overline{M}$). We let γ range over M and $\overline{M}$, and let ψ range over sequences of γ, representing low-level traces. For each local implementation process, these transitions define the messages that may be sent, and the messages that may be received and accepted as genuine. We assume that every message M implements a single high-level communication α. (Intuitively, M is a wire format for α, carrying additional information.) We write $\rho(\gamma)$ for the corresponding high-level communication α, obtained by parsing γ.

The relations $\vdash_{\mathcal{C}}$ model the capabilities of an adversary that controls all machines outside $\mathcal{C}$ to produce (or forge) a message M after receiving (or eavesdropping) the set of messages $\mathcal{M}$. For instance, if the implementation relies on digital signatures, the definition of $\vdash_{\mathcal{C}}$ may reflect the capability of signing arbitrary messages on behalf of the non-compliant machines.

Example 2. We may implement the request-response protocol with processes $A_0 = \overline{a} \,|\, b$, $A_1 = a.\overline{b}$ by re-using the syntax of specification processes and labels (i.e. M is just α), with initial implementation processes (A_0, A_1), implementation transitions

$$A_0 \xrightarrow{\overline{0\ a\ 1}}_0 b \xrightarrow{1\ b\ 0}_0 \mathbf{0} \qquad\qquad A_1 \xrightarrow{0\ a\ 1}_1 \overline{b} \xrightarrow{\overline{1\ b\ 0}}_1 \mathbf{0}$$

and an adversary that can replay intercepted messages and produce any messages from compromised principals, modelled with the deduction rules

$$\frac{}{\mathcal{M}, M \vdash_{\mathcal{C}} M} \qquad\qquad \frac{i \notin \mathcal{C} \qquad i, j \in \{0, 1\} \qquad x \in \{a, b\}}{\mathcal{M} \vdash_{\mathcal{C}} i\ x\ j}$$

Distributed semantics For a given distributed implementation $\langle \widetilde{Q}, \widetilde{\rightarrow}, \vdash_{\mathcal{C}} \rangle$ and set of compliant machines $\mathcal{C}$, we define a global implementation semantics that collects all message exchanges between compliant machines on the network. In our model, compromised processes do not have implementations; instead, the environment represents an abstract adversary with the capabilities specified by $\vdash_{\mathcal{C}}$. Thus, the global implementation transitions $\xrightarrow{\gamma}_{\mathrm{I}}$ relate tuples of compliant implementation processes $\widehat{Q}$ and also collects all compliant messages $\mathcal{M}$ exchanged on the network (starting with an empty set), with the following two rules:

(SendI)
$$\frac{Q_i \xrightarrow{\overline{M}}_i Q_i' \qquad (Q_k = Q_k')^{k \neq i} \qquad i \in \mathcal{C} \qquad \rho(\overline{M}) = i\ a\ j}{\mathcal{M}, \widehat{Q} \xrightarrow{\overline{M}}_{\mathrm{I}} \mathcal{M} \cup M, \widehat{Q'}}$$

(ReceiveI)
$$\frac{Q_i \xrightarrow{M}_i Q_i' \qquad (Q_k = Q_k')^{k \neq i} \qquad i \in \mathcal{C} \qquad \mathcal{M} \vdash_{\mathcal{C}} M \qquad \rho(M) = j\ a\ i}{\mathcal{M}, \widehat{Q} \xrightarrow{M}_{\mathrm{I}} \mathcal{M}, \widehat{Q'}}$$

Rule (SendI) simply records the message sent by a compliant participant, after ensuring that its index i matches the sender recorded in the message interpretation $i\ a\ j$ (so that no compliant participant may send a message on behalf of another). Rule (ReceiveI) enables a compliant participant to input any message M that the adversary may produce from $\mathcal{M}$ (including, for instance, any intercepted message) if its index matches the receiver i in its interpretation $j\ a\ i$; this does not affect $\mathcal{M}$, so the adversary may a priori replay M several times. The wire format M may include additional information, such as a nonce to detect message replays, or some evidence of a previously-received message. The example below illustrates the need for both mechanisms.

Example 3. In the specification $A_0 = \overline{a} \,|\, \overline{a}, A_1 = a \,|\, a$, machine A_1 must discriminate between a replay of A_0's first message and A_0's genuine second message, so these two messages must have different wire formats. For instance, an implementation may include a sequence number, or a fresh nonce.

In the specification $A_0 = \overline{a}, A_1 = a.\overline{b}, A_2 = b$, the implementation of A_2 that receives b from A_1 must check that A_1 previously received a from A_0. For instance, the two messages may be signed by their senders, and A_1's message may also include A_0's message, as evidence that its message on b is legitimate.

Concrete threat model and adequacy The distributed implementations of Definition 1 are expressed in terms of processes and transitions, rather than message handlers for a concrete, cryptographic wire format. (We refer to prior work for sample message formats and protocols for sequential specifications [3,1].)

In the implementation, the use of cryptographic primitives such as digital signatures can prevent the adversary to forge arbitrary messages. On the other hand, since the adversary controls the network, he can resend any intercepted messages and, when it controls a participant, he can at least send any message that this participant could send if it were honest. Accordingly, for any realistic implementation, we intend that the two rules displayed below follow from the definition of $\vdash_{\mathcal{C}}$. We say that an implementation is *adequate* when these two rules are valid.

$$\mathcal{M}, M \vdash_{\mathcal{C}} M \qquad \frac{i \notin \mathcal{C} \qquad Q_i \xrightarrow{\psi}_i Q_i' \xrightarrow{\overline{M_0}}_i Q_i'' \qquad \mathcal{M} \vdash_{\mathcal{C}} M \text{ for each } M \in \psi}{\mathcal{M} \vdash_{\mathcal{C}} M_0}$$

In the second rule, if (1) the adversary controls machine i; (2) a compliant implementation Q_i can send the message M_0 after a trace ψ has been taken; and (3) every message received in ψ can be constructed from $\mathcal{M}$ by the adversary; then he can also construct the message M_0 from $\mathcal{M}$.

Soundness and completeness We first formally relate high-level traces of the specification, ranged over by φ, which include the communications of all processes, to low-level implementation traces, ranged over by ψ, which record only the inputs and outputs of the compliant processes. We then define our property of soundness, stating that every implementation run corresponds to a run of the specification; and our property of completeness, stating that every specification run corresponds to a run of the implementation. These properties depend only on the implementation and specification traces: trace-equivalent specifications would accept the same sound and complete implementations. The correspondence between high-level and low-level traces is captured by the following definition of *valid explanations*.

Definition 2. *A high-level trace $\varphi = \alpha_0 \dots \alpha_{p-1}$ is a* valid explanation *of a low-level trace $\psi = \gamma_0 \dots \gamma_{q-1}$ for a given set $\mathcal{C}$ when there is a partial function ι from (indexes of) low-level messages γ_k of ψ to (indexes of) high-level communications $\alpha_{\iota(k)}$ of φ such that $\rho(\gamma_k) = \alpha_{\iota(k)}$ for $k \in 0..q-1$ and*

1. *the restriction of ι on the indexes of the low-level inputs of ψ is a one-to-one, increasing function to the indexes of the high-level communications of φ with honest receivers (i a j with $j \in \mathcal{C}$);*
2. *the restriction of ι on the indexes of the low-level outputs of ψ is a partial one-to-one function to the indexes of the high-level communications of φ with honest senders (i a j with $i \in \mathcal{C}$); and*
3. *whenever a low-level input precedes a low-level output, their images by ι are in the same order when defined.*

The definition relates every trace of messages sent and received by honest implementations to a global trace of communications between specification processes (including compromised processes). In particular, the specification trace may have additional communications between compromised processes. The relation guarantees that the implementation messages are received in the same order

as the communications in the specification trace. Conversely, since the adversary controls the network, the relation does not guarantee that all low-level outputs are received, or that they are received in order.

For example, consider an implementation that uses specification messages as wire format (that is, M is just α) and let $\mathcal{C} = \{0; 2\}$. We may reflect the trace

$$\psi = (\overline{0\ b\ 2}).(\overline{0\ c\ 1}).(1\ d\ 2).(\overline{0\ a\ 1}).(0\ b\ 2)$$

using, for instance, the valid explanation $\varphi_1 = (0\ c\ 1).(1\ d\ 2).(0\ b\ 2)$ and the index function ι from ψ to φ_1 defined by $\{0 \mapsto 2, 1 \mapsto 0, 2 \mapsto 1, 4 \mapsto 2\}$. Following Definition 2, we check that

1. the restriction of ι to the indexes of the low-level inputs is $\{2 \mapsto 1; 4 \mapsto 2\}$ and relates the inputs of compliant participants in ψ and in φ;
2. the restriction of ι to the indexes of the low-level outputs is $\{0 \mapsto 2; 1 \mapsto 0\}$ and relates two of the three outputs in ψ to the outputs of compliant participants in φ, out of order; and
3. $(1\ d\ 2)$ precedes $(\overline{0\ a\ 1})$ in ψ, but ι is undefined on that low-level output.

Another valid explanation is $\varphi_2 = (1\ d\ 2).(0\ a\ 1).(0\ c\ 1).(0\ b\ 2)$.

We are now ready to define our main security property, which states that an implementation is *sound* when the compliant machines cannot be driven into an execution disallowed by the global specification.

Definition 3 (Soundness). *$\langle \widetilde{Q}, \widetilde{\rightarrow}, \vdash_{\mathcal{C}} \rangle$ is a sound implementation of $\widetilde{P}$ when, for every $\mathcal{C} \subseteq 0..n-1$ and every implementation trace $\emptyset, \widehat{Q} \xrightarrow{\psi}_{\mathrm{I}} \mathcal{M}, \widehat{Q'}$, there exists a source trace $\widetilde{P} \xrightarrow{\varphi}_{\mathsf{S}} \widetilde{P'}$ where φ is a valid explanation of ψ.*

Also, an implementation is *complete* if, when all machines comply, every trace of the global specification can be simulated by a trace of the implementation.

Definition 4 (Completeness). *$\langle \widetilde{Q}, \widetilde{\rightarrow}, \vdash_{\mathcal{C}} \rangle$ is a complete implementation of $\widetilde{P}$ when, for $\mathcal{C} = 0..n-1$ and for every source traces $\widetilde{P} \xrightarrow{\varphi}_{\mathsf{S}} \widetilde{P'}$, there exists an implementation trace $\emptyset, \widetilde{Q} \xrightarrow{\psi}_{\mathrm{I}} \mathcal{M}, \widetilde{Q'}$ where φ is a valid explanation of ψ.*

We easily check that, with our definition of valid explanations, an implementation that is both sound and complete also satisfies the message transparency property discussed in Section 1.

4 Implementing two-party specifications (application)

We instantiate our definitions to specifications with only two participants, such as a client and a server. Such specifications have been much studied using session types. For this section, we set $n = 2$ and implement specifications of the form $\widetilde{P} = (P_0, P_1)$. For simplicity, we also exclude local communications: for any action a, each P_i for $i = 0, 1$ may include either a or $\overline{a}$, but not both.

A simple (insecure) implementation An implementation that re-uses the syntax of specification processes and labels with initial implementation processes (P_0, P_1) (that is, Q_i is just P_i, and M is just α) is generally not sound. Consider, for instance, the specification $A_0 = a.\overline{e} \,|\, b.(\overline{e} + \overline{c})$ and $A_1 = \overline{a} \,|\, \overline{b} \,|\, e \,|\, c$. After the communications a, e, and b, the implementation of machine 1 would be in state c, while that of machine 0 would be in state $\overline{e} + \overline{c}$: machine 1 should accept a message c. However, after the communications b, e, and a, the implementation of machine 1 would still be in state c, but that of machine 0 would be in state $\overline{e}$, unable to send c: machine 1 should not accept a message c. In state c machine 1 does not know whether a message c from machine 0 is legitimate or not. Therefore, an implementation accepting the message c is unsound, and an implementation refusing it is incomplete.

History-tracking Implementations Our implementation relies on a refinement of the specification syntax and semantics to keep track of past communications: local processes are of the form $P : \psi$ where ψ is a sequence of global communications, each tagged with a fresh nonce ℓ used to prevent message replays (any received message whose tag already occurs in ψ is ignored).

- We use $P_0 : \varepsilon$ and $P_1 : \varepsilon$ as initial processes (where ε is the empty sequence).
- We define local implementation transitions $\rightarrow_i$ from the initial specification $\widetilde{P}$ and the specifications traces:

$$\frac{\widetilde{P} \xrightarrow{\rho(\psi)}_{\mathsf{S}} \widetilde{P'} \xrightarrow{i\ a\ j}_{\mathsf{S}} \widetilde{P''} \qquad i\ a\ j\ \ell \notin \psi}{P'_i : \psi \xrightarrow{\overline{i\ a\ j\ \ell}}_i P''_i : \psi.(i\ a\ j\ \ell)} \qquad \frac{\widetilde{P} \xrightarrow{\rho(\psi)}_{\mathsf{S}} \widetilde{P'} \xrightarrow{i\ a\ j}_{\mathsf{S}} \widetilde{P''} \qquad i\ a\ j\ \ell \notin \psi}{P'_j : \psi \xrightarrow{i\ a\ j\ \ell}_j P''_j : \psi.(i\ a\ j\ \ell)}$$

  where i, j is either $0, 1$ or $1, 0$ and where ρ yields a specification trace by erasing all nonces ℓ in ψ.
- We define the adversary knowledge $\vdash_{\mathcal{C}}$ by $\dfrac{}{\mathcal{M}, M \vdash_{\mathcal{C}} M}$ and $\dfrac{i \notin \mathcal{C}}{\mathcal{M} \vdash_{\mathcal{C}} i\ a\ j\ \ell}$.

Hence, an action is locally enabled only when it extends the specification trace recorded so far and the nonce ℓ is fresh. The adversary may send a message either by eavesdropping it or by constructing it with a compromised sender. (Pragmatically, a concrete implementation may generate ℓ at random, or increment a message sequence number, and may use a more compact representation of ψ.)

Theorem 1. $\langle (P_0 : \varepsilon, P_1 : \varepsilon), \widetilde{\rightarrow}, \vdash_{\mathcal{C}} \rangle$ *is a sound and complete implementation of* (P_0, P_1).

The soundness of our implementation above relies on every machine recording every communication (since it is either sending or receiving every message); this approach does not extend to specifications with more than two machines, inasmuch as these machines do not directly observe actions between two remote machines.

5 Implementability by typing

In the preceding section, we presented a complete and sound implementation for binary sessions. We now illustrate some difficulties in the general n-ary case.

Example 4. Consider a variant of Example 1 with the same V_1, V_2, V_3 and R but with the election officer E split into $E_1 = c_1.c_1.\overline{r_1}$ and $E_2 = c_2.c_2.\overline{r_2}$. One of the voters (say, V_1) may cheat, and send *both* c_1 to E_1 and c_2 to E_2. To prevent this attack, E_1 and E_2 would need to communicate with one another, thereby breaking message transparency. Therefore no adequate implementation of this example can be both sound and complete. To prevent this pattern, we will demand that both sides of a sum affect the same participants in the same order.

Example 5. Consider now the specification

$$A = (\overline{b_1} \,|\, \overline{c_1}) + (\overline{b_2} \,|\, \overline{c_2}) \qquad B = b_1 \,|\, b_2 \qquad C = c_1 \,|\, c_2$$

The process A can send either b_1 to B and c_1 to C, or b_2 to B and c_2 to C. A dishonest machines in charge of running A can send b_1 to B and c_2 to C. Therefore no adequate implementation of this example can be both sound and complete. To prevent this pattern, we will also demand that both sides of some parallel composition (e.g.under a sum) affect the same participants in the same order.

To prevent these situations, we develop a type system with two kinds of types, for sequential processes and (possibly) parallel processes:

$\sigma ::=$	sequential types	$\pi ::=$	parallel types
$\mathbf{0}$	completion	σ	sequential type
$i.\sigma$	sequence	$i.\pi$	sequence
		$\pi \,|\, \pi$	parallel

Intuitively, our types indicate (by their indexes) which other participants may be affected by each action, and in what order. In Example 1, action c_1 would be of type 3.4 since it is received by process 3 and this reception may contribute to the emission of r_1 to process 4.

We define subtyping with three base rules and two context rules:

$$\mathbf{0} \leq \sigma \qquad \pi \,|\, \pi \leq \pi \qquad \pi \leq \pi \,|\, \pi \qquad \frac{\pi \leq \pi'}{i.\pi \leq i.\pi'} \qquad \frac{\pi_1 \leq \pi_1' \quad \pi_2 \leq \pi_2'}{\pi_1 \,|\, \pi_2 \leq \pi_1' \,|\, \pi_2'}$$

Thus, for a sequential type, we can "forget" potential future actions and obtain a less precise type and, for parallel types, we can duplicate or merge parallel copies carrying the same information.

We type local processes at each machine $i \in 0..n-1$, in a given environments Γ that map channels to parallel types. The typing judgment $\Gamma \vdash_i P : \pi$ indicates that P can be given type π at machine i in environment Γ, with the rules below:

$$\text{(Send)}\quad \Gamma, a:\pi \vdash_i \overline{a} : i.(\pi\backslash i) \qquad \text{(Receive)}\quad \frac{\Gamma, a:\pi \vdash_i P : \pi' \quad \pi' \leq \pi}{\Gamma, a:\pi \vdash_i a.P : \pi'} \qquad \text{(Sub)}\quad \frac{\Gamma \vdash_i P : \pi \quad \pi \leq \pi'}{\Gamma \vdash_i P : \pi'}$$

$$\text{(Nil)}\quad \Gamma \vdash_i \mathbf{0} : i \qquad \text{(Plus)}\quad \frac{\Gamma \vdash_i P_0 : \sigma \quad \Gamma \vdash_i P_1 : \sigma}{\Gamma \vdash_i P_0 + P_1 : \sigma} \qquad \text{(Par)}\quad \frac{\Gamma \vdash_i P_0 : \pi \quad \Gamma \vdash_i P_1 : \pi'}{\Gamma \vdash_i P_0 \,|\, P_1 : \pi \,|\, \pi'} \qquad \text{(Repl)}\quad \frac{\Gamma \vdash_i P : \pi}{\Gamma \vdash_i !P : \pi}$$

where $\pi\backslash i$ is π after erasure of every occurrence of i.

Rule (Send) gives to the output $\overline{a}$ the type of action a (minus i) preceded by i. This records that $\overline{a}$ at host i affects any process that receives on a. Conversely, rule (Receive) gives to $a.P$ the type of the continuation process P, and checks that it is at least as precise as the type of action a. Rule (Sub) enables subtyping. Rule (Nil) gives type i to an empty process, since it has no impact outside i. Rule (Plus) ensures that the two branches of a choice have the same effect, a sequential type, excluding e.g.the typing of the specifications in Example 4. Rules (Par) and (Repl) deal with parallel compositions.

For instance, in the environment $\Gamma = r_1 : 4, r_2 : 4, c_1 : 3.4, c_2 : 3.4$, the processes of Example 1 have types

$$\Gamma \vdash_0 V_1 : 0.3.4 \quad \Gamma \vdash_1 V_2 : 1.3.4 \quad \Gamma \vdash_2 V_3 : 2.3.4 \quad \Gamma \vdash_3 E : 3.4 \quad \Gamma \vdash_4 R : 4$$

Conversely, the processes $V_1 = V_2 = V_3 = \overline{c_1} + \overline{c_2}$ are not typable within the unsafe specification of Example 4, because $\overline{c_1}$ and $\overline{c_2}$ necessarily have incompatible types.

We end this section by defining typability for global specifications, with a shared environment for all machines and a technical condition to ensure consistency on channels with parallel types.

Definition 5. *A global specification $\widetilde{P}$ is* well-typed *when, for some environment Γ and each $i \in 0..n-1$, we have $\Gamma \vdash_i P_i : \pi_i$ and, for each $(a : \pi) \in \Gamma$, either π is (a subtype of) a sequential type, or $\widetilde{P}$ has at most one reception on a.*

6 History-tracking implementations

In this section we present an implementation for session specifications. We prove that the implementation is complete, and that it is sound when the specification is well-typed (Definition 5). The resulting family of implementations subsumes those presented in the special cases of binary sessions (Section 4) and sequential sessions (Section 7).

Multiparty specifications and history-tracking implementations As seen in Example 1, in a multiparty system, a local action at one machine may causally depend on communications between other machines. To avoid cheating, we embed evidence of past execution history in our implementation messages. Thus, to implement Example 1, the code for the election officer E explicitly forwards evidence of receiving c_1 twice in order to convince R that it can send the result r_1.

As a preliminary step, we enrich processes with histories of prior communications. Then, we equip these processes with a refined semantics, with rules that

define how histories are collected and communicated. Finally, the presence of histories allows us to constrain each local implementation by prescribing what messages may be sent and received at runtime. (Our history-tracking implementation is related to locality semantics for CCS; for instance Boudol and Castellani [2] use *proved labelled transitions* that keep track of causality by recording where each action occurs in a process.)

Histories are lists of messages, defined by the following grammar:

$H ::=$	Histories
$\quad \varepsilon$	$\quad$ empty history
$\quad H.M$	$\quad$ recorded receive
$M ::=$	Messages
$\quad (H\ i\ a\ j\ \ell)$	

Each message $H\ i\ a\ j\ \ell$ records an action a between sender i and receiver j (where i and j are indexes of processes in the global specification), with a history H that provides evidence that action a is indeed enabled. In addition, ℓ denotes a unique nonce, freshly generated for this message, used to avoid replays.

The syntax of processes extended with histories is as follows:

$T ::=$	Threads
$\quad \mathbf{0}$	$\quad$ inert thread
$\quad \overline{a}$	$\quad$ asynchronous send
$\quad a.P$	$\quad$ asynchronous receive
$\quad P + P'$	$\quad$ choice
$\quad !P$	$\quad$ replication
$R ::=$	History-tracking processes
$\quad (T_0 : H_0 \mid T_1 : H_1 \mid \ldots \mid T_{k-1} : H_{k-1})$	
	$\quad$ parallel composition of history-tracking threads
$\widetilde{R} ::=$	Global history-tracking specifications
$\quad (R_0, R_1, \ldots, R_{n-1})$	$\quad$ tuple of n history-tracking processes

where P ranges over the local processes of Section 2. Our specification processes are split into different parallel components, each with its own history. For example, when $P = a.\overline{b} \mid \overline{c}$ receives a, this receive enables action $\overline{b}$ (and is tracked in its history) but is independent from action $\overline{c}$. So, a *thread* T is a (specification) process without parallel composition at top-level, a *history-tracking process* R is a collection of threads in parallel, each with its history, and a *global history-tracking specification* $\widetilde{R}$ is a tuple of n history-tracking processes.

The function $\mathit{Ths}(P : H)$ normalizes the process P into a parallel composition of threads, each annotated with the same history H. It is recursively defined from $\mathit{Ths}(P_0 \mid P_1 : H) = \mathit{Ths}(P_0 : H) \mid \mathit{Ths}(P_1 : H)$. Further, the function $[\![\cdot]\!]_0$ normalizes a global specification, with an initial, empty history. Conversely, since any thread is a local process, a history-tracking process (resp. a global history specification) stripped of its histories is a local process (resp. a global specification).

Semantics of history specifications ($\rightarrow_{\mathsf{h}}$ and $\rightarrow_{\mathsf{H}}$) We define labelled transitions for history specifications. We write $R \xrightarrow{\gamma}_{\mathsf{h}} R'$ when R can evolve to R' with action γ. It corresponds to an input or an output on one of its threads.

$$\text{(SendH)}\quad \frac{T \xrightarrow{\overline{a}}_{\mathsf{P}} P}{T : H \xrightarrow{\overline{H\ i\ a\ j\ \ell}}_{\mathsf{h}} P : H} \qquad \text{(ReceiveH)}\quad \frac{T \xrightarrow{a}_{\mathsf{P}} P}{T : H' \xrightarrow{H\ i\ a\ j\ \ell}_{\mathsf{h}} P : H'.(H\ i\ a\ j\ \ell)}$$

$$\text{(ParH)}\quad \frac{R \xrightarrow{\gamma}_{\mathsf{h}} R'}{R \,|\, R'' \xrightarrow{\gamma}_{\mathsf{h}} R' \,|\, R''}$$

Rule (ReceiveH) records the message in the thread history. In contrast, rule (SendH) does not record the message, since our semantics is asynchronous. Rule (ParH) is a rule for parallel contexts; we omit the symmetric rule.

We write $\widetilde{R} \xrightarrow{M}_{\mathsf{H}} \widetilde{R}'$ to represent communications between history-tracking specifications, with a single global rule:

$$\frac{R_i \xrightarrow{\overline{H\ i\ a\ j\ \ell}}_{\mathsf{h}} R_i^\circ \quad (R_k = R_k^\circ)^{k\neq i} \qquad R_j^\circ \xrightarrow{H\ i\ a\ j\ \ell}_{\mathsf{h}} R_j' \quad (R_k^\circ = R_k')^{k\neq j} \quad H\ i\ a\ j\ \ell \notin \widetilde{R}}{\widetilde{R} \xrightarrow{H\ i\ a\ j\ \ell}_{\mathsf{H}} \widetilde{R}'}$$

A communication step consists of a send (at machine i) followed by a receive (at machine j) for local history-tracking processes (possibly with $i = j$). The condition $H\ i\ a\ j\ \ell \notin \widetilde{R}$ excludes multiple usage of the same message.

Example 6. Consider a global specification with three processes $\overline{a}$, $a.\overline{b}$ and b; the following is a trace of its global history specification.

$$\begin{aligned}[\![(\overline{a}, a.\overline{b}, b)]\!]_0 &\xrightarrow{\varepsilon\ 0\ a\ 1}_{\mathsf{H}} (\mathbf{0} : \varepsilon, \overline{b} : \varepsilon\ 0\ a\ 1, b : \varepsilon) \\ &\xrightarrow{(\varepsilon\ 0\ a\ 1)1\ b\ 2}_{\mathsf{H}} (\mathbf{0} : \varepsilon, \mathbf{0} : \varepsilon\ 0\ a\ 1, \mathbf{0} : (\varepsilon\ 0\ a\ 1)1\ b\ 2)\end{aligned}$$

Local semantics ($\rightarrow_i$) We are now ready to define distributed implementation transitions locally, for each machine $i \in 0..n-1$, written $R_i \xrightarrow{M}_i R_i'$:

$$\frac{[\![\widetilde{P}]\!]_0 \xrightarrow{\psi}_{\mathsf{H}} \widetilde{R'} \quad H\ i\ a\ j\ \ell \notin \widetilde{R'} \qquad R_i' \xrightarrow{\overline{H\ i\ a\ j\ \ell}}_{\mathsf{h}} R_i^\circ \quad (R_k' = R_k^\circ)^{k\neq i} \qquad R_j^\circ \xrightarrow{H\ i\ a\ j\ \ell}_{\mathsf{h}} R_j'' \quad (R_k^\circ = R_k'')^{k\neq j}}{R_i' \xrightarrow{\overline{H\ i\ a\ j\ \ell}}_i R_i^\circ} \qquad \frac{[\![\widetilde{P}]\!]_0 \xrightarrow{\psi}_{\mathsf{H}} \widetilde{R'} \quad H\ i\ a\ j\ \ell \notin \widetilde{R'} \qquad R_i' \xrightarrow{\overline{H\ i\ a\ j\ \ell}}_{\mathsf{h}} R_i^\circ \quad (R_k' = R_k^\circ)^{k\neq i} \qquad R_j^\circ \xrightarrow{H\ i\ a\ j\ \ell}_{\mathsf{h}} R_j'' \quad (R_k^\circ = R_k'')^{k\neq j}}{R_j^\circ \xrightarrow{H\ i\ a\ j\ \ell}_j R_j''}$$

These rules (with identical premises) prescribe that a distributed implementation can send or receive a message when the corresponding communication is enabled in some global history specification state that is reachable from the initial history specification process $[\![\widetilde{P}]\!]_0$.

A naive concrete implementation may enumerate all possible runs at every communication. A more efficient implementation would cache this computation and perform incremental checks, or perform this computation at compile-time. (See [3,1] for optimized implementations in the sequential case.)

Distributed implementation We finally define our distributed implementation, with $\widetilde{Q} = [\![\widetilde{P}]\!]_0$ as initial implementation processes, with $(\xrightarrow{\gamma}_i)_i$ defined above as transition relations between implementation processes, and with capabilities $\vdash_{\mathcal{C}}$ for the adversary defined by

$$\mathcal{M}, M \vdash_{\mathcal{C}} M \qquad \frac{(\mathcal{M} \vdash_{\mathcal{C}} M_m)^{m<k} \qquad i \notin \mathcal{C}}{\mathcal{M} \vdash_{\mathcal{C}} M_0.\cdots.M_{k-1}\ i\ a\ j\ \ell}$$

The first rule states that the adversary can eavesdrop messages on the network. The second rule states that the adversary can build any message sent by a dishonest participant, with a history recursively composed of sequence of messages previously obtained. Conversely, the adversary cannot forge any message from a compliant machine (i.e.a machine $i \in \mathcal{C}$). This can be cryptographically enforced by authenticating messages and countersigning their histories. Our implementation is adequate, in particular a dishonest participant can behave as an honest participant.

(The global transition rules $\rightarrow_{\mathrm{I}}$ for our distributed implementation follow from the general definitions of Section 3.)

Soundness and completeness Our implementation is complete, that is, it can simulate any specification trace:

Theorem 2 (Completeness). *$\langle [\![\widetilde{P}]\!]_0, \widetilde{\rightarrow}, \vdash_{\mathcal{C}} \rangle$ is a complete implementation of $\widetilde{P}$.*

Our main result states that our implementation is also sound *when applied to well-typed specifications*; as explained in Section 5, many other specifications cannot be safely implemented.

Theorem 3 (Soundness by Typing). *If $\widetilde{P}$ is well-typed, then $\langle [\![\widetilde{P}]\!]_0, \widetilde{\rightarrow}, \vdash_{\mathcal{C}} \rangle$ is sound.*

7 Sequential multiparty sessions (application)

We consider secure implementations of sequential multiparty sessions, as defined by Corin *et al.* [3]. Their sessions are a special case of process specifications. We recall their grammar, which defines a session as a parallel composition of *role processes*, each process specifying the local actions for one role of the session.

$\tau ::=$	Payload types
int | string	base types
$p ::=$	Role processes
$!(f_i : \widetilde{\tau}_i\ ;\ p_i)_{i<k}$	send
$?(f_i : \widetilde{\tau}_i\ ;\ p_i)_{i<k}$	receive
$\mu\chi.p$	recursion declaration
χ	recursion
0	end
$S ::=$	Sequential session (with n roles)
$(r_i = p_i)_{i \in 0..n-1}$	

Their role processes must alternate between send and receive actions, and moreover only the initiator role process (p_0) begins with a send. Thus, there is always at most one role that can send the next message, as expected of a sequential session. From a more global viewpoint, a session is represented as a directed graph, whose nodes represent roles and whose arrows are indexed by unique communication labels. The paths in the graph correspond to the global execution traces for the session. Given an additional implementability property on these paths, named "no blind fork", they construct a cryptographic implementation that guarantees *session integrity*, even for sessions with dishonest participants, a property closely related to Soundness (Definition 3).

To illustrate our approach, we show that our generic implementation directly applies to every session that they implement (although with less compact message formats). We translate their role processes into our syntax as follows:

$$\begin{aligned}
[\![p]\!] = {} & \textstyle\sum_{i<k}(\overline{f_i}) \qquad \text{when } p = !(f_i : \widetilde{\tau_i}\ ;\ p_i)_{i<k} \\
& \mid \textstyle\prod_{(?(f_i:\widetilde{\tau_i}\ ;\ p_i)_{i<k})\in p,\ i<k,\ p_i=!(f'_j:\widetilde{\tau'_j}\ ;\ p'_j)_{j<l}} !f_i.\textstyle\sum_{j<l}(\overline{f'_j}) \\
& \mid \textstyle\prod_{(?(f_i:\widetilde{\tau_i}\ ;\ p_i)_{i<k})\in p,\ i<k,\ p_i=\chi,\ (\mu\chi.!(f_j:\widetilde{\tau_j}\ ;\ p_j)_{j<l})\in p} !f_i.\textstyle\sum_{j<l}(\overline{f'_j}) \\
[\![(r_i = p_i)_{i\in 0..n-1}]\!] = {} & ([\![p_i]\!])_{i\in 0..n-1}
\end{aligned}$$

Each node in a session graph has an input arrow and one or several output arrows, representing an internal choice between outputs. Accordingly, our translation associates to each node a replicated input (using $q \in p$ to denote any syntactic subprocess q of p) following by a choice between asynchronous outputs. In addition, the initial role for the session is an internal choice between outputs, translated to an internal choice of asynchronous outputs. By induction on paths in the graph, we can check that our translation behaves as the initial sequential session. (The sequentiality of the session follows from the presence of a single choice between outputs in every reachable state, so we can replicate all inputs, whether they occur in recursive loops or not.)

For any given sequential session, typability of the translation (Definition 5) coincides with the "no blind fork" property [3]. Hence, every sequential session supported by their implementation is typable, and can also be implemented in our general framework:

Theorem 4. *The history-tracking implementation of the translation of a sequential session that respects the "no blind fork" property is sound and complete.*

8 Conclusions

We have given an account of distributed specifications and their implementations in three steps: (1) a global specification language; (2) a distributed implementation semantics; (3) correctness and completeness results, depending on an implementability condition. In combination, this yields a general framework for designing and verifying n-ary communication abstractions with strong, guaranteed security properties. (In comparison, the work on sequential multiparty sessions [3] can now be seen as a specialized cryptographic implementation for the sequential case.)

Future work Each of the above contributions may be improved. First, the specification language may be extended, for example by accounting for the message contents and stating additional security goals (secrecy, causality, commitment) or by adding mobile channel names. Second, the implementation semantics may be further refined. Although we believe our implementation approach is fully general, its performance can clearly be improved, for example by avoiding redundant communications of history once it is either irrelevant or common knowledge. Also, even if the cryptographic protection mechanisms are standard, their efficient implementation remains delicate. More experimentally, we have not prototyped an actual session compiler for our implementation scheme, and it remains unclear how to deal efficiently e.g.with infinite numbers of states. This leads us to our third point, possible improvement on implementability conditions: typability only provides a sufficient condition; we have built an efficient (quadratic) typability verifier, and our only examples of specifications that are sound but not typable can be easily rewritten into typable ones, but still it would be interesting to address this gap.

Acknowledgements Karthikeyan Bhargavan, Pierre-Malo Deniélou, Gurvan le Guernic, Cosimo Laneve, James Leifer, Jean-Jacques Lévy, Eugen Zălinescu, and the anonymous reviewers provided useful comments on this work.

References

1. K. Bhargavan, R. Corin, P.-M. Deniélou, C. Fournet, and J. J. Leifer. Cryptographic protocol synthesis and verification for multiparty sessions. In *22nd IEEE Computer Security Foundations Symposium (CSF'09)*, July 2009.
2. G. Boudol and I. Castellani. Flow models of distributed computations: three equivalent semantics for CCS. *Information and Computation*, 114(2):247–314, 1994.
3. R. Corin, P.-M. Deniélou, C. Fournet, K. Bhargavan, and J. J. Leifer. A secure compiler for session abstractions. *Journal of Computer Security (Special issue for CSF'07)*, 16(5):573–636, 2008.
4. D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.
5. S. J. Gay and M. Hole. Types and subtypes for client-server interactions. In *Programming Languages and Systems, 8th European Symposium on Programming (ESOP)*, pages 74–90, 1999.
6. K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In *Programming Languages and Systems, 7th European Symposium on Programming (ESOP)*, volume 1381, pages 22–138. Springer, 1998.
7. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008*, pages 273–284. ACM, 2008.
8. R. Milner. *Communication and Concurrency*. Prentice-Hall, Inc., 1989.