J. Log. Algebraic Methods Program. 152 (2027) 101186

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

journal homepage: www.elsevier.com/locate/jlamp

# Partial order semantics and operational semantics for strand spaces, via Rocq

Matteo Busi [a,*], Riccardo Focardi [a], Joshua Guttman [b], Flaminia Luccio [a]

[a] Università Ca' Foscari, Venice, Italy
[b] Zeno's Arrow Consulting, Newton, MA, USA

ARTICLE INFO

*Keywords:*
Strand spaces
Security protocols
Rocq

ABSTRACT

Strand spaces are a special-purpose theory for security protocols, i.e. tiny distributed programs that use cryptography for authentication, confidentiality, and related goals. Strand spaces yield compact and understandable proofs that security protocols meet their goals—and counterexamples when they do not—as well as supporting an automated search tool that can compute the security goals a protocol satisfies.

The strand space notion of execution, the *bundle*, is partially ordered, where the causal dependence of receptions on transmissions generates the ordering.

In this paper, we examine how this partially ordered semantics relates to sequential, trace-oriented notions of execution. We provide a Structured Operational Semantics for protocols in strand spaces. We prove that every SOS history yields a bundle that respects its causal relations, and that—given a bundle—there is an ordering of its events that yields an SOS history. Our results are formalized in Rocq using the StrandsRocq framework.

## 1. Introduction

Security protocols are distributed activities in which principals interact via messages to establish authentication, confidentiality, and other goals. While some events may be causally independent, others exhibit causal dependencies. For example, a message reception depends causally on the transmission that makes this message available; otherwise the reception event could not occur. Similarly, when a protocol-compliant principal takes a step in a session, this depends on any previous steps that the protocol defines; otherwise a compliant principal would not be ready to take the step.

However, this causal ordering of dependency is a partial order. For many pairs of events, neither depends on the other, and these events could have happened in either order. A server, for example, may require two principals to send messages before it can provide its service, while remaining indifferent to which message is transmitted first. In asynchronous network models, it may even be unable to determine which was sent first. Hence, events that contribute to a protocol session may still be causally incomparable. Thus, a partial order semantics is a natural modeling choice for the fundamental semantics of security protocols, and respects their intrinsically concurrent structure.

The partial order semantics is also practically desirable for analyzing them. If one would like to explore their possible executions systematically, it is more efficient to collapse the irrelevant interleavings in which causally unrelated events occur in different orders into a single structure—called a *bundle*—that represents only the causal partial order.

* Corresponding author.
*E-mail addresses:* matteo.busi@unive.it (M. Busi), focardi@unive.it (R. Focardi), joshua.guttman@gmail.com (J. Guttman), luccio@unive.it (F. Luccio).

https://doi.org/10.1016/j.jlamp.2026.101186
Received 15 June 2026; Received in revised form 24 September 2026; Accepted 25 September 2026
Available online 27 September 2026
2352-2208/© 2026 The Authors. Published by Elsevier Inc. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

On the other hand, one would also like a systematic way to generate executions step-by-step, and this requires an operational semantics in which events occur sequentially.

The goal of this paper is to connect these two semantic views of security protocol behavior. In particular, we prove that every execution of the operational semantics determines a bundle (Theorem 4). Conversely, every bundle may be generated stepwise in the structured operational semantics (Theorem 5). Indeed, unless an event $e_2$ causally precedes another event $e_1$ in the bundle, there is a run of the structured operational semantics in which $e_1$ occurs before $e_2$ (Theorem 6). Thus, the bundle accurately summarizes the ordering constraints on the runs of the operational semantics.

Beyond reconciling different semantics views, this paper takes a step toward a syntactic integration. We show how the syntax used in Cryptographic Protocol Shapes Analyzer (CPSA) [1] specifications can be naturally embedded in StrandsRocq [2] and equipped with an operational semantics. This bridges automated protocol analysis of CPSA with the manual, mechanized approach of StrandsRocq, enabling protocols previously analyzed in CPSA to be analyzed directly in StrandsRocq. This way, each tool can also serve as a cross-validator for the other. This work shares with ALVIE [3] the broader goal of connecting automated analysis with trustworthy formal models: while ALVIE learns behavioral models directly from black-box implementations, here we connect CPSA-style protocol analysis with an explicit, mechanically checked semantics in Rocq.

We have formalized and proved all of our claims using the Rocq theorem prover, based on the initial StrandsRocq formalization [2]. Some auxiliary lemmas and proof scripts were developed with the help of AI-assisted tools and were subsequently reviewed, edited, and checked by the authors in Rocq.

Rocq distinguishes systematically between assertion-like entities that are propositional in nature, such as formulas and the predicates that yield formulas when applied to arguments, and value-like entities, such as concrete values and the functions or higher-order entities that yield concrete values when applied to their arguments. Computation is supported in Rocq by the latter but not the former. A strength of our formalization is that our results are available on both sides of this divide. Using the computational version, we can extract the (partially ordered) bundle that a (linear) run generates, or find a (linear) run that yields a given (partially ordered) bundle.

*What is not in this paper.* We will avoid discussing how to use the strand space theory to prove protocols correct, as that has been the subject of a long series of papers, dating back to the 1990s, e.g. [4–6]; new techniques continue to be added [2]. A strength of the strand ideas is that they have suggested strong methods for reasoning about protocols compositionally [2,7–9], and even provide good heuristics for protocol design [10]. Strands are also well mechanized; the CPSA tool is polished and usable [1], and robust enough to verify complex protocols with hardware dependencies [11].

Indeed, an additional contribution of this paper is to forge a link between the syntax and methods of CPSA and the details of the StrandsRocq theory (Section 4); this link creates the opportunity to verify the steps CPSA takes mechanically in StrandsRocq, or to generate protocol-specific lemmas in StrandsRocq that justify CPSA in pruning its search.

*Artifact availability.* The artifact accompanying this paper, including all Rocq proofs and formalizations, is archived on Zenodo with a permanent DOI at https://doi.org/10.5281/zenodo.22939512.

*To Giangi!* We warmly thank Giangi for the semantic tradition he helped weave as part of the Pisa school of semantics, from concurrency and mobility to process calculi and global computing. This paper is connected to that tradition by more than one strand: through Flaminia's Pisa roots, Matteo's supervisors from Pisa, Riccardo's supervisory lineage from Pisa via Bologna, and Joshua's warm memories of Giangi from visits to Pisa.

## 2. Background

This section provides a few examples, as well as informal descriptions of central strand space notions.

### 2.1. Some security protocols

Security protocols serve a variety of purposes. For instance, many protocols are intended to establish a new shared secret—which can be used to protect a subsequent conversation—between two participants, in which case ensuring that each of the participants authenticates the identity of the peer may be a crucial part of the goal. The *TLS handshake* in its "mutual" or "bilateral" version is a centrally important protocol of this kind. In other cases, one participant may not need to know who the peer is; the unilateral TLS handshake is pervasively used in these cases, e.g. when a server is willing to set up a cryptographically protected conversation with any client, but the client wants to ensure the conversation is with the intended server. For instance, the conversation may culminate in the client paying the server: The server may be willing to accept payment from anyone, while the client wants to ensure her payment goes to a server that can be trusted to ship a purchased item to her home.

By contrast, the *TLS record layer* is a protocol that uses a shared secret—typically having been set up by the TLS handshake—to ensure confidentiality and integrity for a sequence of messages, and their in-order delivery.

*Terminology.* Asymmetric encryption (also known as public key encryption) uses a pair $\mathsf{pk}(A)$, $\mathsf{dk}(A)$ to protect a plaintext. The former is made public, so that any participant wanting to protect a message $m$ for $A$ solely uses $\mathsf{pk}(A)$ to generate an encrypted message we will write as $\{|m|\}_{\mathsf{pk}(A)}$. $A$ keeps $\mathsf{dk}(A)$ secret, and uses it when needed to decrypt such messages to retrieve $m$.

A *nonce* is simply a bitstring a principal chooses so as to be able to recognize messages belonging to the same session. When selected in a sufficiently unpredictable way, it is called *fresh* and *random*.

We write $m_1, m_2$ to represent a pair of messages.

**Example 1** (The Yes-Or-No Protocol). Suppose that an *answerer* $A$ has published a public encryption key $\mathsf{pk}(A)$. *Questioners* would like to be able to deliver questions to the answerer and receive an answer which will be either *yes* or *no*. The goal of the protocol is to ensure that if an adversary does not know the answerer's private decryption key $\mathsf{dk}(A)$, she cannot tell what the answer was, even if she guesses correctly what the question was.[1]

To accomplish this, the questioner selects the intended question $Q$, and freshly generates two random nonces we will call $y$ and $n$. The questioner transmits the encrypted message $\{|Q, y, n|\}_{\mathsf{pk}(A)}$. If the questioner subsequently receives the first nonce $y$ back, then the answer was *Yes*. If she receives the second nonce $n$, then the answer was *No*.

The answerer's behavior is dual. Given an encrypted message $m$, $A$ uses the private decryption key $\mathsf{dk}(A)$ to find if $m$ is of the form $\{|Q, y, n|\}_{\mathsf{pk}(A)}$. If it is not, or if $Q$ is not a yes-or-no question, the answerer discards $m$. If it is, the answerer determines the correct answer. If this is *Yes*, the answerer sends the first nonce $y$, destroying the second one $n$. If the answer is *No*, the answerer does the reverse.

An adversary that knows what $Q$ is being asked in $m$ still learns nothing. That's because the random nonces that we have called $y$ and $n$ are simply random bitstrings. We call them $y$ and $n$ as a hint to how they are used in the protocol. But they could just as easily have been chosen in the opposite order. In that case, observing the same bitstring would have conveyed the opposite information to the questioner. Hence the adversary who cannot tell which order they appeared in the encryption $\{|Q, y, n|\}_{\mathsf{pk}(A)}$ learns nothing. In any secure encryption scheme, the ciphertexts $\{|Q, y, n|\}_{\mathsf{pk}(A)}$ and $\{|Q, n, y|\}_{\mathsf{pk}(A)}$ are indistinguishable to any observer who does not possess $\mathsf{dk}(A)$.

The protocol illustrates how to use meaningless strings like $y$ and $n$ to obtain useful information from a protocol-compliant peer; also, how to incorporate branching behaviors in a protocol. The questioner also learns that the answer is recent: It cannot have been transmitted until after $y$ and $n$ were randomly chosen.

**Example 2** (A Key Distribution Protocol). Suppose that two peers $A$ and $B$ want to have an encrypted conversation, but they have no public keys set up and share no secrets with each other currently. However, let us suppose there is also a *key server* $S$, and $A$ and $B$ have both established shared keys with $S$ that they can use for symmetric encryption. We will call these secrets $\mathsf{ltk}(A, S)$ and $\mathsf{ltk}(B, S)$ resp., and assume that the adversary does not have access to them. Our protocol, while based on a classic protocol by Rafael Yahalom [12–14], is new in this paper; we adapted it so natural executions have pairs of incomparable events.

In this *Concurrent Yahalom* protocol, the server first receives a message $A, B, N_a$ with the name of the first participant, the name of the second participant, and a bitstring serving as a nonce, presumably from the first participant $A$. It then receives a message $A, B, N_b$, presumably from the second participant $B$. The server prepares a fresh random key $K$ and transmits it separately in two messages, one encrypted for $A$ and one for $B$. These take the forms

$$\{|A, B, K, N_a, N_b|\}_{\mathsf{ltk}(A,S)} \quad \text{and} \quad \{|A, B, K, N_a, N_b|\}_{\mathsf{ltk}(B,S)},$$

in which only the encryption key $\mathsf{ltk}(\cdot, S)$ differs. The server does not know whether the initial messages came from the expected participants, but the encryptions should render these messages harmless unless the participants intended to communicate.

The behaviors of the two peers are very similar. The first peer $A$, to get a key shared with $B$, selects a bitstring $N_a$ and sends $A, B, N_a$, expecting to receive a message of the form $\{|A, B, K, N_a, N_b|\}_{\mathsf{ltk}(A,S)}$. To confirm possession of $K$, it sends $\{|\mathsf{tag}_1, A, B, N_a, N_b|\}_K$, expecting to receive a confirmation of $B$'s possession of the form $\{|\mathsf{tag}_2, A, B, N_a, N_b|\}_K$.

The second principal $B$, to get a key shared with $A$, places its own name $B$ second in $A, B, N_b$; expects to receive $\{|A, B, K, N_a, N_b|\}_{\mathsf{ltk}(B,S)}$; and confirms key possession by sending $\{|\mathsf{tag}_2, A, B, N_a, N_b|\}_K$, expecting to receive $\{|\mathsf{tag}_1, A, B, N_a, N_b|\}_K$.

Protocol analysis assuming $\mathsf{ltk}(A, S), \mathsf{ltk}(B, S)$ uncompromised shows, that despite an active adversary:

1. A session key $K$ transmitted by the server is never compromised;
2. A full run of $A$ as first participant with a nonce $N_a$ that was successfully freshly generated implies matching runs of
   - the server generating $K$ and
   - the second participant $B$, whose final reception is not confirmed;
3. A full run of $B$ as second participant with a nonce $N_b$ that was successfully freshly generated implies matching runs of
   - the server generating $K$ and
   - the first participant $A$, whose final reception is not confirmed.

The guarantees (2,3) that $A$ and $B$ receive are entirely symmetrical, as the similarity of their roles suggests.

We use CPSA for this analysis, described in Section 2.3. CPSA's justification for claim 2 establishes that every full execution of the first participant with fresh $N_a$ contains at least the information, redrawn manually for clarity, shown in Fig. 1. The analysis also shows corresponding actions of the two peers are independent and may happen in either order. They transmit their first request independently in either order; they receive the key packages in either order; they transmit their key confirmations in either order; and they receive each other's confirmation in either order (if both occur). Thus, there are in fact sixteen different sequential orderings in which a successful global run may occur. A similar result is found for claim 3. The non-compromise result (claim 1) holds because no execution contains a run of the server protocol with session key $K$ and also a reception in which $K$ is obtained without cryptographic protection, assuming $\mathsf{ltk}(A, S), \mathsf{ltk}(B, S)$ are used only as the protocol requires.

[1] See [9] for a more thorough discussion of the yes-or-no protocol.

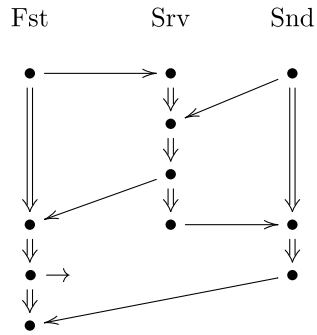



**Fig. 1.** Concurrent Yahalom: an execution.

### 2.2. Strands, bundles, protocols

We work within the strand space framework [4,15]. In this framework, each local session of a single principal engaging in a protocol yields a (usually very short) sequence of transmissions and receptions, and each of these sequences is called a *strand*, such as each of the vertical columns in Fig. 1. The basic actions of the adversary may be regarded as strands too, although no adversary strands are present in Fig. 1. Now fix a set of messages or terms $\mathcal{A}$ and a set of objects $\Sigma$ that we will call *strands*. Their structure does not matter in this paper. When $t \in \mathcal{A}$, we use $+t$ to express the event that $t$ is transmitted, and $-t$ to express it being received.

**Definition 1** (Strand preliminaries)**.**

1. A *directed message* is an object $+t$ or $-t$ where $t \in \mathcal{A}$.
2. A *trace* function is a map $\mathsf{tr}$ from $\Sigma$ to lists of directed messages.
3. If $tr$ is a trace function, then the pair $(\Sigma, tr)$ is a *strand space*. Fix a strand space $(\Sigma, tr)$.
4. The *length* of a strand $s \in \Sigma$ is the length of the list $\mathsf{tr}(s)$. We write $|s|$ for the length of $s$.
5. A *node* $n$ is a pair $n = (s, i)$ where $s \in \Sigma$ and $0 \le i < |s|$. When $n = (s, i)$, we let $\mathsf{strand}(n) = s$ and $\mathsf{idx}(n) = i$.
6. The *directed message at* node $n = (s, i)$, written $\mathsf{dmsg}(n)$, is the (zero-based) $i^{\text{th}}$ entry in the list $\mathsf{tr}(s)$.
7. A node is *positive* iff $\mathsf{dmsg}(n) = +t$ for some $t$, and *negative* iff $\mathsf{dmsg}(n) = -t$ for some $t$. We write $\mathsf{pos}(n)$ and $\mathsf{neg}(n)$ resp. in these two cases.
8. The *communication edges* are the pairs of nodes $(n_1, n_2)$ such that, for some $t \in \mathcal{A}$, $\mathsf{dmsg}(n_1) = +t$ and $\mathsf{dmsg}(n_2) = -t$. We write $n_1 \to n_2$ when $(n_1, n_2)$ is a communication edge.
9. The *strand succession edges* are the pairs of nodes $(n_1, n_2)$ such that, for some $s \in \Sigma$ and $i$, $\mathsf{strand}(n_1) = \mathsf{strand}(n_2)$ and $\mathsf{idx}(n_1) + 1 = \mathsf{idx}(n_2)$. We write $n_1 \Rightarrow n_2$ when $(n_1, n_2)$ is a strand succession edge.

Thus, $\mathsf{pos}(\cdot)$ and $\mathsf{neg}(\cdot)$ partition the nodes, the former being nodes on which the transmissions occur and the latter being those on which receptions occur.

An execution consists of a collection of nodes that have occurred on this occasion, together with edges of the two types. One type is the *communication edges* $n_1 \to n_2$ that connect a transmission node $n_1$ to a reception node $n_2$ that receives the same message. Communication can happen across different strands or on the same strand, and happens asynchronously. They are shown as the horizontal or diagonal edges in Fig. 1. Our definitions allow the case in which $n_1 \to n_2$ and the nodes lie on the same strand. We are modeling an asynchronous communication mechanism, as is generally the case for security modeling, so this could make sense. We can also have repeated reception, in a broadcast-like way in which both $n_1 \to n_2$ and $n_1 \to n_3$.

The *strand succession edges* $n_1 \Rightarrow n_2$ connect one node to the next node occurring on the same strand, i.e. when $n_1 = (s, i)$ and $n_2 = (s, i+1)$ for some $s$ and $i$. Strand succession edges are shown vertically in Fig. 1. We say that $n_2$ *depends directly on* $n_1$ if either $n_1 \to n_2$ or $n_1 \Rightarrow n_2$. In the first case, the transmission $n_1$ explains how $n_2$ was able to receive the message on that occasion. In the second case, $n_1$ explains how the principal became ready to execute $n_2$.

**Definition 2** (Bundle). If $\mathcal{N}$ is a finite set of nodes and $\to_{\mathcal{B}}, \Rightarrow_{\mathcal{B}} \subseteq \mathcal{N} \times \mathcal{N}$, we say that $\mathcal{B} = (\mathcal{N}, \to_{\mathcal{B}}, \Rightarrow_{\mathcal{B}})$ is a *bundle* iff

1. $\Rightarrow_{\mathcal{B}}$ and $\to_{\mathcal{B}}$ consist of strand-succession and communication edges resp., i.e.:
   (a) $n_1 \Rightarrow_{\mathcal{B}} n_2$ and $n_1 = (s, i)$ implies $n_2 = (s, i+1)$;
   (b) $n_1 \to_{\mathcal{B}} n_2$ implies $n_1$ transmits some message $m$ and $n_2$ receives $m$;
2. if $n_2$ is a reception node, there is exactly one $n_1 \in \mathcal{N}$ such that $n_1 \to_{\mathcal{B}} n_2$;
3. if $n_2 = (s, i+1) \in \mathcal{N}$, then $n_1 \Rightarrow_{\mathcal{B}} n_2$ where $n_1 = (s, i)$; and finally
4. the transitive closure $(\to_{\mathcal{B}} \cup \Rightarrow_{\mathcal{B}})^+$ is acyclic.

Fig. 1 shows a bundle. Our definition allows the same transmission to be received more than once, so our communication model is a sort of broadcast. It also allows a reception to occur at a subsequent time, so it is asynchronous. Neither of these properties is seen in Fig. 1. However, if the Snd strand is lengthened to include the final reception that the role permits, and it receives the message transmitted on the third node of the Fst strand, that makes the asynchrony explicit.

We write the transitive closure $(\rightarrow_{\mathcal{B}} \cup \Rightarrow_{\mathcal{B}})^+$ as $\prec_{\mathcal{B}}$ and the reflexive transitive closure $(\rightarrow_{\mathcal{B}} \cup \Rightarrow_{\mathcal{B}})^*$ as $\preceq_{\mathcal{B}}$. Finiteness and acyclicity together imply that $\preceq_{\mathcal{B}}$ is a well-founded relation. Thus, bundles satisfy a form of well-founded induction, namely:

Every non-empty set $S \subseteq \mathcal{N}$ has $\preceq_{\mathcal{B}}$-minimal elements.

This "bundle induction" principle provides the fundamental way to reason about what can happen in a strand space, and it is causal in a thoroughgoing way. That is, the $\preceq_{\mathcal{B}}$-minimal elements of $S$ are the events at which $S$ has first been caused to hold, and protocol analysis is about finding the precursor events that caused them to occur. A number of strand space papers have developed increasingly sophisticated consequences of the fundamental bundle induction principle, which lead to very effective protocol analysis methods [2,4,5,9,15].

We use $\rightarrow$ and $\Rightarrow$ generically (Definition 1, Clauses 8–9), writing $n_1 \rightarrow n_2$ when the directed message of $n_1$ is some $+m$ and that of $n_2$ is $-m$, and writing $n_1 \Rightarrow n_2$ when $n_1 = (s, i)$ and $n_2 = (s, i+1)$ for some $s$ and $i$. We use $\rightarrow_{\mathcal{B}}, \Rightarrow_{\mathcal{B}}$ for the subsets of these relations in a particular bundle $\mathcal{B}$. However, when we think no confusion will result, we omit the $\mathcal{B}$ subscript and use $\rightarrow$ and $\Rightarrow$ for the relations in $\mathcal{B}$.

We typically regard the *messages* as forming an algebra generated over a set of sorted variables by operations such as message pairing, encryption, hashing, digital signature, and the like.

The *adversary* has a set of strands modeling her ability to manipulate these messages, typically based on their algebraic structure. She can: Encrypt a message she receives using a key she receives; Decrypt an encrypted message if she receives the right key, thereby obtaining its plaintext; Pair messages together or Separate (unpair) them; Guess basic values; and so on.

A *protocol* is a finite set of strands, called the *roles* of the protocol, over these messages. In the Concurrent Yahalom protocol, these are the First, Second, and Server roles. The *strand space* of a given protocol is the set of all instances we obtain by sort-respecting instantiations of the variables occurring in the roles by messages, together with the adversary strands.

### 2.3. Cryptographic protocol shapes analyzer

CPSA, the Cryptographic Protocol Shapes Analyzer, carries out protocol analysis in the strand space model [1,15]. A protocol is specified by giving some strands with parameters; this determines the strand space consisting of all of their instances, together with the standard adversary strands. Some additional information may also be specified. For instance, in Concurrent Yahalom, we would like to assume that every run of the server generates a fresh random session key.

One can then run CPSA on a sequence of queries about that protocol. A *query* is essentially the antecedent of a security claim such as each of the claim (1)–(3) in Section 2.1. CPSA must characterize the situations in which this query may be satisfied. Specifically, the queries in Section 2.1 say:

1. Suppose that the server $S$ has a run with parameters $A, B, K$ for the names of the first participant, second participant, and the session key, and suppose that $\mathsf{ltk}(A, S)$ and $\mathsf{ltk}(B, S)$ are uncompromised (i.e. used only in accordance with the protocol); suppose also that the session key $K$ is observed as a message available to the adversary.
2. Suppose that $A$ has a full run in the role of first participant with server $S$ and intended second participant $B$; suppose also that its parameter $N_a$ was freshly generated and $\mathsf{ltk}(A, S)$ and $\mathsf{ltk}(B, S)$ are uncompromised.
3. Suppose that $B$ has a full run in the role of second participant with server $S$ and intended first participant $A$; suppose also that its parameter $N_b$ was freshly generated and $\mathsf{ltk}(A, S)$ and $\mathsf{ltk}(B, S)$ are uncompromised.

In each case, CPSA starts from a minimal structure satisfying the assumptions mentioned in the query. When the structure has message receptions that cannot be explained via adversary activity from previous transmissions (subject to the freshness and non-compromise assumptions), CPSA considers all relevant additions to the structures [6]. The search branches when different additions could contribute to alternative explanations. If the search terminates, CPSA returns the set of all minimal, essentially different structures, called *shapes*, that extend the starting structure and in which all of the receptions are explained [15].

Each analysis determines a formula. It is an implication, whose antecedent is the starting query. Its conclusion describes the resulting shapes. The conclusion is the disjunction of the descriptions of each resulting shape. This formula is a *strongest security goal* achieved by the protocol with the given query as antecedent [9,16]. If the set of shapes is the empty set—the initial query is unsatisfiable given the protocol, as arises in query (1)—then the conclusion is simply *falsehood*, which is the disjunction of 0 possibilities.

CPSA can also be given a security goal of the same form, and it will determine whether that is satisfied.

### 2.4. Strands Rocq: key ideas

StrandsRocq [2] started with a very faithful formalization of an analysis from the first strand space paper [4]. Having done that, the authors recognize that a variety of new types of analysis can be carried out in the same framework. For instance, a protocol that uses keys subject to a key usage policy can be analyzed using protocol reasoning together with the deductive properties of the key usage policy. StrandsRocq led to another exciting opportunity also, namely to vary the adversary model. The StrandsRocq development led to an attractive principle, related to the honest ideals of the original strand paper [4], which identifies message invariants that—if the adversary respects them—suffice to establish protocol correctness for a protocol $\Pi_1$. Sometimes an alliance of the standard Dolev-Yao adversary [17] and another protocol $\Pi_2$ still respects this invariant. This yields a compositional correctness condition for $\Pi_1$: Its correctness holds even for adversaries that can marshal those protocols $\Pi_2$ that do not falsify these invariants.

$$\frac{}{\mathsf{IndBundle}(\emptyset,\emptyset,\emptyset)}\ \textsc{Empty} \qquad \frac{\mathsf{IndBundle}(N,L,C) \quad n \notin N \quad \mathsf{pos}(n) \quad \mathsf{idx}(n)=0}{\mathsf{IndBundle}(N \cup \{n\}, L, C)}\ \textsc{Send0}$$

$$\frac{\mathsf{IndBundle}(N,L,C) \quad p \in N \quad n \notin N \quad \mathsf{pos}(n) \quad p \Rightarrow n}{\mathsf{IndBundle}(N \cup \{n\}, L \cup \{(p,n)\}, C)}\ \textsc{Send}$$

$$\frac{\mathsf{IndBundle}(N,L,C) \quad q \in N \quad n \notin N \quad q \to n \quad \mathsf{idx}(n)=0}{\mathsf{IndBundle}(N \cup \{n\}, L, C \cup \{(q,n)\})}\ \textsc{Recv0}$$

$$\frac{\mathsf{IndBundle}(N,L,C) \quad q \in N \quad p \in N \quad n \notin N \quad q \to n \quad p \Rightarrow n}{\mathsf{IndBundle}(N \cup \{n\}, L \cup \{(p,n)\}, C \cup \{(q,n)\})}\ \textsc{Recv}$$

**Fig. 2.** Rules generating bundles inductively.

In this paper, we sharpen definitions somewhat, including the fundamental notion of bundle, while retaining the previous results. We also remodularized the development so that it is easy to apply the theory across different algebras of messages.

## 3. Formalizing bundles

We formalize Definition 2 directly in StrandsRocq.[2] The definition concerns a triple $\mathcal{B} = (\mathcal{N}, \to, \Rightarrow)$ with $\to, \Rightarrow \subseteq \mathcal{N} \times \mathcal{N}$; the formalization represents this via a record type `bundle_type` with three components: a list of nodes, a list `inter` of communication edges, and a list `intra` of strand-succession edges. We often access them via Rocq's `ListSet` interface, which eases using the lists to represent finite sets. The predicate `is_bundle` then states when elements of type `bundle_type` satisfy the bundle conditions of Definition 2: the two edge relations have the intended strand-space meanings, receptions have unique incoming communication edges, strands are prefix-closed, and the causal graph is acyclic. Thus, `is_bundle` is the Rocq counterpart of Definition 2.

**Example 3** (A running example). Fig. 1 satisfies `is_bundle`. Its nodes are the bullets occurring on the three strands, its intra-strand edges are the vertical succession edges, and its communication edges are the horizontal or diagonal arrows from transmissions to receptions. The two initial request transmissions by the first and second participants are incomparable: neither one depends on the other, so either may occur first in a sequential history. By contrast, a reception node requires a prior matching transmission, and a non-initial node on a strand requires its immediate predecessor on that strand. The bundle must therefore record the causal constraints though it may leave causally unrelated events unordered.

However, although the graph in Fig. 1 can be viewed extensionally as a partial order, it could also be generated step by step. The first participant's initial transmission may be added before the second participant's, or conversely; later nodes may be added whenever their immediate causes have already been added. Different construction orders may produce the same partial-order bundle.

Moreover, the step-by-step construction is useful for formal reasoning. It is possible, though tedious, to prove directly that a concrete graph such as Fig. 1 satisfies `is_bundle`: for that particular choice of node and edge lists, one must prove each extensional condition separately. The proof must show that every listed edge has the right strand-space meaning, that all endpoints are listed as nodes, that each reception has exactly one communication source, that strand predecessors are present, and that the resulting causal graph is acyclic. Instead, the inductive characterization below enables ordered bundle construction: nodes are added one at the time as they are causally justified and extensional bundle properties are recovered via a general (*soundness*) theorem.

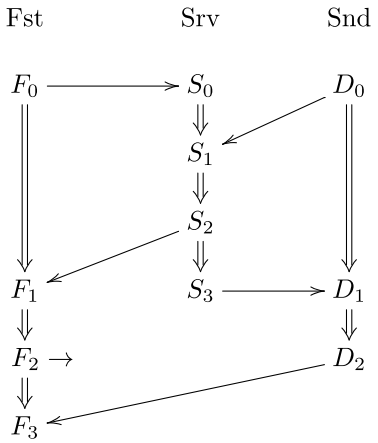
### 3.1. An inductive characterization

We now introduce the inductive characterization of bundles.[3] The relation $\mathsf{IndBundle}(N, L, C)$ holds when the finite graph with nodes $N$, intra-strand edges $L$, and communication edges $C$ is an *inductive bundle* according to the rules in Fig. 2.

Notice that $p \Rightarrow q$ and $p \to q$ are the generic properties of the nodes $p$ and $q$, independent of any bundle $\mathcal{B}$. Specifically, $p \to q$ iff $\mathsf{pos}(p)$ and $\mathsf{neg}(q)$ and they both carry the same message, and $p \Rightarrow q$ iff $\mathsf{idx}(q) = \mathsf{idx}(p) + 1$ and $\mathsf{strand}(p) = \mathsf{strand}(q)$. The Empty rule is the base case and trivially says that the empty graph is an inductive bundle. The remaining rules each extend an existing bundle with a fresh node $n \notin N$, differing only in what edges must be present or are added. Rules Send0 and Send handle transmissions: since a transmission needs no incoming communication edge, neither rule modify the edges in $C$. The difference between the two rules is that Send0 requires $\mathsf{idx}(n) = 0$, so $n$ is the first event on its strand and no intra-strand edge is added in $L$, while Send requires a strand predecessor $p \Rightarrow n$ already in $N$ and adds the intra-strand edge $(p, n)$ to $L$. Neither rule can be applied twice to the same $n$, as one premise is $n \notin N$.

[2] See the file `Common/Bundles.v` in the artifact.

[3] See `Common/BundleInductive.v` in the artifact for the full mechanization. We will use $\mathsf{pos}(n)$ and $\mathsf{neg}(n)$ to express the partition of nodes into positive and negative, i.e. transmissions and receptions. We use $\mathsf{idx}(n)$ for its index, i.e. $i$ if $n = (s, i)$, in which case $\mathsf{strand}(n) = s$.




(1) EMPTY provides the empty bundle; then add:
(2) $F_0$ by SEND0, since it is an initial transmission;
(3) $D_0$ by SEND0, for the same reason;
(4) $S_0$ by RECV0, using the communication edge $F_0 \to S_0$;
(5) $S_1$ by RECV, using both $S_0 \Rightarrow S_1$ and $D_0 \to S_1$;
(6) $S_2$ by SEND, using the strand edge $S_1 \Rightarrow S_2$;
(7) $S_3$ by SEND, using the strand edge $S_2 \Rightarrow S_3$;
(8) $F_1$ by RECV, using both $F_0 \Rightarrow F_1$ and $S_2 \to F_1$;
(9) $F_2$ by SEND, using the strand edge $F_1 \Rightarrow F_2$;
(10) $D_1$ by RECV, using both $D_0 \Rightarrow D_1$ and $S_3 \to D_1$;
(11) $D_2$ by SEND, using the strand edge $D_1 \Rightarrow D_2$;
(12) $F_3$ by RECV, using both $F_2 \Rightarrow F_3$ and $D_2 \to F_3$.

**Fig. 3.** Concurrent Yahalom bundle and one inductive derivation of it.

Dually, RECV0 and RECV handle receptions: every reception must be justified by a transmitter $q \in N$ with $q \to n$, so both rules add the communication edge $(q, n)$ to $C$. RECV0 again handles the case $\mathsf{idx}(n) = 0$, where no intra-strand edge is needed, while RECV additionally requires a strand predecessor $p \Rightarrow n$ and adds $(p, n)$ to $L$.

The Rocq constructors have the same shape as the rules: `G_empty` for the empty graph, `G_pos_zero` and `G_pos` for transmissions, and `G_neg_zero` and `G_neg` for receptions. The formal definition uses lists rather than mathematical sets, so the constructors add new nodes and edges at the head of the corresponding lists. This is a representation choice: the intended object is still the finite graph, not the particular list order.

It is important that this definition does not replace the partial order by a total order. Instead, a derivation of $\mathsf{IndBundle}(B)$ is a causal construction order for the bundle. If two nodes are incomparable in the bundle, there will be inductive derivations of it in which each occurs earlier, as with the two initial participant transmissions in Fig. 1.

**Example 4** (Generating the bundle in Fig. 1)**.** The left half of Fig. 3 repeats Fig. 1 with node labels. The first participant strand contains $F_0, \dots, F_3$ from top to bottom; the server strand contains $S_0, \dots, S_3$; and the second participant strand contains $D_0, \dots, D_2$.

The communication edges shown in Fig. 3 are

$$F_0 \to S_0, \quad D_0 \to S_1, \quad S_2 \to F_1, \quad S_3 \to D_1, \quad D_2 \to F_3.$$

The transmission $F_2$ does not have a communication edge: there is no corresponding reception in this bundle, so the transmission indicator at the bottom of the figure is not a $\to$ edge. One inductive derivation of this bundle is shown in the right half of Fig. 3. It is only one possible derivation. For instance, the two initial transmissions $F_0$ and $D_0$ may be added in either order, and after the server transmissions are present, the receptions $F_1$ and $D_1$ may be ordered in either way compatible with their local predecessors. The inductive construction constructs the graph in some linear order, but the generated object is still the partial-order bundle that is the same as Fig. 1.

### 3.2. Soundness

The first theorem says that the inductive rules generate only bundle graphs that satisfy `is_bundle`.

**Theorem 1** (Soundness of inductive bundles)**.** *If* $\mathsf{IndBundle}(N, L, C)$*, then* $(N, L, C)$ *is a bundle.*

In the Rocq development this theorem is `IndBundle_sound`.

**Proof.** We proceed by induction on the derivation of $\mathsf{IndBundle}(N, L, C)$ and verify the clauses of Definition 2. The EMPTY case is immediate. In every other case, the induction hypothesis gives all the bundle properties for the old graph. The premises of the constructor state that each newly listed edge has the appropriate strand-space meaning and that its old endpoints are already nodes; its other endpoint is the fresh node being added. Thus all edge endpoints are nodes. The SEND and RECV rules add precisely the immediate strand edge required for a non-initial node, whereas the index-zero rules require no such edge. Hence strand-prefix closure is preserved.

It remains to consider communication sources and acyclicity. A send constructor introduces no communication edge and its fresh positive node creates no new reception obligation. A receive constructor adds one communication edge from the matching transmission to the fresh negative node. This is the unique source of that reception: an old communication edge cannot already target the fresh node, since every endpoint of an old edge is an old node. The unique sources of old receptions are unchanged.

Finally, every edge added by a constructor is directed from an old node to the fresh node. Moreover, no old edge is incident to the fresh node, again by freshness and the endpoint property supplied by the induction hypothesis. The fresh node is therefore maximal in the enlarged causal graph. A cycle containing it would have to leave it, which is impossible; a cycle not containing it would be a cycle in the old graph, contradicting the induction hypothesis. All bundle conditions follow. □

### 3.3. Proving results about all bundles

Bundles $\mathcal{B}$ have a natural notion of size, namely the cardinality of the set of nodes $|nodes(\mathcal{B})|$. Since this determines a well-founded order on bundles, we can prove[4] an induction principle based on it:

**Theorem 2** (Bundle size induction)**.** *Let* $\Phi$ : `bundle_type` → `Prop` *be a property on the type of bundles. To prove for all* $\mathcal{B}$*, if* $\mathcal{B}$ *is a bundle then* $\Phi(\mathcal{B})$*, it suffices to prove:*

*For all* $\mathcal{B}$ *and integers* $k$*, if*

1. $\mathcal{B}$ *is a bundle with* $|nodes(\mathcal{B})| \leq k$*, and*
2. $\Phi(C)$ *holds for all bundles* $C$ *of smaller size* $|nodes(C)| < k$*,*

*then it follows that* $\Phi(\mathcal{B})$*.*

**Proof.** By induction on $|nodes(\mathcal{B})|$. □

*Maximal nodes, called sinks.* When reasoning about all bundles, it is helpful to be able to identify maximal nodes in any given bundle, and the effect of removing them.

A *sink* of a bundle $\mathcal{B}$ is a node $m$ with no other node causally after it:

$$m \in nodes(\mathcal{B}) \quad \text{and} \quad \forall n \in nodes(\mathcal{B}).\ n \neq m \Rightarrow \neg(m \preceq_{\mathcal{B}} n).$$

If $\mathcal{B}$ is a non-empty bundle, then some node in $\mathcal{B}$ is a sink of $\mathcal{B}$. This holds because a sink is a minimal node relative to the order that is converse to $\preceq_{\mathcal{B}}$, and acyclicity ensures there are nodes that are minimal relative to the converse of $\preceq_{\mathcal{B}}$.

If $\mathcal{B}$ is a bundle and $m$ is a sink, then *removing* $m$ from $\mathcal{B}$ is the structure containing the nodes other than $m$ and the intra-strand and inter-strand edges of $\mathcal{B}$ that do not lead to $m$. By the definition of sink, no edges lead out of $m$ in $\mathcal{B}$.

**Lemma 1** (Bundle sink removal)**.**

1. *Removing a sink from a non-empty bundle yields a smaller structure, i.e. one with fewer nodes.*
2. *The result of removing a sink from a non-empty bundle is a bundle.*

*An induction principle based on sinks.* By the preceding, we can specialize Theorem 2 to use sink removal to obtain a particular bundle smaller than a given non-empty bundle. In the following statement we assume that the nodes of bundles are presented without duplications.

**Corollary 1** (Bundle sink induction)**.** *Let* $\Phi$ : `bundle_type` → `Prop` *be a property on the type of bundles. To prove for all* $\mathcal{B}$*, if* $\mathcal{B}$ *is a bundle then* $\Phi(\mathcal{B})$*, it suffices to prove:*

*For all* $\mathcal{B}$*, if, for all nodes* $m \in nodes(\mathcal{B})$*:*

1. $\mathcal{B}$ *is a bundle,*
2. $m$ *is a sink node of* $\mathcal{B}$*,*
3. $\Phi$ *holds of the result of removing* $m$ *from* $\mathcal{B}$

*then it follows that* $\Phi(\mathcal{B})$*.*

**Proof.** Apply Theorem 2. For a non-empty bundle, removing a sink preserves the bundle property by Lemma 1 (item 2) and strictly decreases the number of nodes by Lemma 1 (item 1). Thus the induction hypothesis applies to the resulting bundle. For the empty bundle, the premise of the stated induction step holds vacuously. □

### 3.4. Completeness

The converse theorem is the more challenging direction. It says that the inductive rules characterize all standard bundles, up to the irrelevant ordering of the finite lists $N, L, C$ used to represent the sets of nodes, strand succession edges, and communication edges.

**Theorem 3** (Completeness of inductive bundles)**.** *If* $(N, L, C)$ *is a bundle, then there are lists* $N'$*,* $L'$*, and* $C'$ *such that* $N'$*,* $L'$*, and* $C'$ *contain the same elements as* $N$*,* $L$*, and* $C$*, respectively, and* $\mathsf{IndBundle}(N', L', C')$*.*

This is the theorem `IndBundle_complete` in StrandsRocq. The "same elements" clauses are expressed in Rocq by a predicate `SameSet`, because the finite sets are implemented as lists. So the theorem says that every bundle has an inductive presentation.

**Proof.** Because nodes are represented by a list, we first remove duplicate occurrences; this preserves both the represented node set and the bundle property. We then use the induction principle of Cor. 1. If $N$ is empty, the endpoint conditions force both edge sets to be empty as well, up to `SameSet`, and EMPTY applies.

[4] See the file `Common/BundleSizeInduction.v`.

Suppose the bundle is non-empty. Therefore it has a sink $m$. Remove $m$ and every edge whose target is $m$. The result is again a bundle by Lemma 1 (item 2). The node set is strictly smaller by Lemma 1 (item 1), so the induction hypothesis gives lists $N'$, $L'$, and $C'$ representing this smaller graph and therefore an inductive derivation for them.

It remains to reconstruct the original graph. Write $m = (s, i)$ and distinguish its sign and index. If $m$ is positive and $i = 0$, no strand or communication edge can target it, and SEND0 adds it back. If $m$ is positive and $i > 0$, prefix closure supplies its immediate predecessor $(s, i-1)$ and the corresponding strand edge; the predecessor remains in $N'$, so SEND applies. If $m$ is negative, the bundle property supplies its unique matching transmission and communication edge, whose source also remains in $N'$. For $i = 0$ we therefore use RECV0; for $i > 0$, prefix closure additionally supplies the immediate predecessor and we use RECV. The sign and uniqueness properties ensure that these are exactly the edges targeting $m$. Reattaching them therefore produces node and edge lists that are `SameSet` to the original lists. This completes the induction. □

**Example 5.** In the labelled bundle of Fig. 3, the node $F_3$ is a sink of the full bundle: no edge leaves it, and no other node is causally after it. Removing $F_3$ also removes the two edges that target it, namely $F_2 \Rightarrow F_3$ and $D_2 \rightarrow F_3$. The remaining graph is still a bundle. The reception $F_3$ has disappeared, so the transmission $D_2$ no longer needs to justify any reception inside the remaining bundle; likewise $F_2$ no longer has a successor on its strand. Thus both $D_2$ and $F_2$ become sinks in the smaller graph.

At this point there is a genuine choice. If we remove $D_2$ next, then the edge $D_1 \Rightarrow D_2$ is removed and $D_1$ becomes a sink. Removing $D_1$ next removes its incoming edges $D_0 \Rightarrow D_1$ and $S_3 \rightarrow D_1$, after which $S_3$ is also maximal. Alternatively, after removing $F_3$ we could remove $F_2$ first. This removes $F_1 \Rightarrow F_2$, making $F_1$ a sink. These different choices correspond to different ways of peeling off maximal events from the same partial order.

One complete sink-removal order is

$$F_3,\ D_2,\ D_1,\ F_2,\ F_1,\ S_3,\ S_2,\ S_1,\ S_0,\ D_0,\ F_0.$$

Reading this order backwards gives an inductive construction of the original bundle. For example, $F_0$ and $D_0$ are restored by SEND0; $S_0$ is restored by RECV0, using $F_0 \rightarrow S_0$; $S_1$ is restored by RECV, using both $S_0 \Rightarrow S_1$ and $D_0 \rightarrow S_1$; and so on until $F_3$ is restored by RECV, using $F_2 \Rightarrow F_3$ and $D_2 \rightarrow F_3$. Thus the completeness proof turns a declarative bundle into an inductive derivation by repeatedly choosing a sink, removing it, and then reversing the process.

### 3.5. Role in the development

The inductive characterization is useful because it isolates the causal core of bundles from the later details of an operational semantics. The rules above already describe when an event is allowed to be appended to a partial execution: all its immediate causes must be present. The soundness theorem guarantees that every such construction denotes a genuine strand-space bundle. The completeness theorem guarantees that no standard bundle is lost by reasoning in this constructive style.

This is the bridge used in the next sections. The operational semantics will generate linear histories, whereas bundles are partial-order objects. The inductive characterization explains why these views agree: a history is acceptable exactly when it is a linearization of the causal construction of a bundle.

### 3.6. A previous inductive notion

An inductive approach to bundles can also be found in work of Li and Pang [18]. An inductive definition formalized in Isabelle is found in [18, §4.3], to which they unhelpfully give the name `bundles`. It is very similar to the rules we give in Section 3.1.

They claim soundness, like our Theorem 1, namely that their inductive definition yields only objects that are bundles as in Definition 2. This is established in their Lemmas 2–5. There are however rough points. For one thing, their Lemma 3 appears to miss the uniqueness in Definition 2, Clause 2, and may be inconsistent with it. They consider possible sending nodes $n'$ transmitting the same message that $n$ receives, and appear to assert that all such $n'$ are in a bundle that contains $n$.

They appear to have no analogue to our Theorem 3.

Moreover, their $\texttt{edges}(\mathcal{B})$ set is effectively the union $\rightarrow \cup \Rightarrow$, meaning that they do not distinguish whether a pair of nodes is a strand succession edge $n_1 \Rightarrow n_2$ or a communication edge $n_1 \rightarrow n_2$, or perhaps both. A pair of nodes is both kinds of edge when a strand sends a message $m$ at node $i$ and receives it back at node $i+1$, hearing its own echo so to speak. This is less problematic in practice, but leads the authors to assume in various places that the sending strand is distinct from the receiving strand in any communication. In fairness, it is unclear in many strand space papers, e.g. [4], whether the authors regard the edges simply as the ordered pair of the endpoints, or as labeled to distinguish communication edges from strand succession edges. Presumably the latter was meant.

## 4. Protocols

Up until this point, we have worked relative to any given strand space. At this point, we will consider how to define strand spaces to match what can happen in a particular protocol in the presence of an active adversary. Then in Section 5 we will provide a structured operational semantics that derives the possible execution histories for a given protocol. Section 6 then shows that these execution histories represent exactly the bundles over the strand space of that protocol.

```
choice "answerer"
  vars [[A] is Name; [Q; y; n] is Text]
  nonorig []
  uniqorig []
  traces [
      trace [recv {|Q, y, n|}_pk(A)]
      then [
          trace "yes" [send y];
          trace "no" [send n]]].
```

$\rightsquigarrow$

```
defrole "answerer-yes"
  vars [[A] is Name; [Q; y; n] is Text]
  trace [recv {|Q, y, n|}_pk(A); send y]
  nonorig []
  uniqorig [];
defrole "answerer-no"
  vars [[A] is Name; [Q; y; n] is Text]
  trace [recv {|Q, y, n|}_pk(A); send n]
  nonorig []
  uniqorig []
```

**Fig. 4.** Role specification with choice (l), and the choice-free roles it expands to (r).

### 4.1. The roles of a protocol

A *protocol* is defined by two components. First, there is a finite set of *roles*, each one of which is a strand that contains variables, called its *parameters*. Each strand generates a set of *instances*, namely all of the parameter-free "ground" strands that are obtained by applying a valuation $\eta$ that maps the parameters to ground terms.

Second, there may be some *restrictions*, which are essentially assumptions or rules that must be satisfied for an execution to be a legitimate run of the protocol. For instance, one might stipulate that every run of the Server in Concurrent Yahalom generates a fresh random session key. This is essentially requiring that in every run, if $K$ is generated as a session key by a Server instance, then there is no other node in the run where $K$ is transmitted without having been previously received. This is a uniqorig assumption. One may also stipulate nonorig assumptions, saying (e.g.) that if a server's role uses parameter values $A, S$, then the corresponding long-term key $\mathsf{ltk}(A, S)$ is transmitted *nowhere* in the run; though it may be *used* for encryption and decryption, it does not contribute to the plaintext of any transmitted message. Other formulas may also be stipulated as rules [1,9].

In this paper, we will largely ignore the restrictions. Our reason for doing so is that they do not play a part in constructing executions. Rather, they help to determine the security goals that an execution satisfies. If an execution requires too improbable an event, such as two random choices that produce the same result, or a random choice that equals a well-protected long-term secret, we will not count it as a counterexample to a security goal. Thus, each goal is effectively qualified with an implicit "Ignoring events of negligible probability…". Hence, we will regard a protocol as a finite set $P$ of parametric strands $\rho \in P$. In practice, we take them from CPSA input files (Section 4.2).

The *adversary* will also be represented by a finite set of roles (Section 4.3).

### 4.2. CPSA-style protocols

The operational semantics will apply to strand spaces generated by protocols, i.e. finite sets of roles, written in a CPSA-style syntax.[5] Each role being a strand with parameters, the strand space contains all of its instances. These instances are generated in a sort-respecting way, as each variable and ground term is associated with a sort. A valuation $\eta$ is a *sort-respecting* map from variables to ground terms, and the function $\mathsf{instantiate_trace}_\eta$ maps a parametric role trace to a ground trace of directed messages.

In the Rocq development, a role contains a name, a list of sorted variables, a trace of parametric send and receive events, and the CPSA fields nonorig and uniqorig. Concretely, the notation used in the development is close to the surface syntax of CPSA. The answerer behavior in the yes-or-no protocol from Section 2.1, for instance, can be written as a shared-prefix choice, as shown on the left in Fig. 4.

This display should be read as syntactic sugar. The variable declaration says that $A$ is a name, while $Q$, $y$, and $n$ are text values. The answerer first receives the encrypted question $\{|Q, y, n|\}_{\mathsf{pk}(A)}$. At that point the two branches still have the same observed prefix. The ``yes'' branch sends $y$, while the ``no'' branch sends $n$. The Rocq object is a family of role records, and the trace entries are parametric events. The choice notation expands or "desugars" to two ordinary roles with the same variable declarations and the same initial receive event, shown on the right side of Fig. 4. The two expanded roles share the common prefix $[-\{|Q, y, n|\}_{\mathsf{pk}(A)}]$, followed respectively by $+y$ and $+n$. Choice is therefore handled by ordinary role-prefix compatibility, and introduces no additional transition rule. Only after a valuation instantiates the variables do these traces become the ground events (directed messages) used by the operational semantics.

[5] For readability, CPSA syntax appears in lowercase in this paper; in the Rocq mechanization it is uppercase.

### 4.3. Penetrator roles

The same protocol representation can also be used for adversary behavior. The penetrator protocol $P_{\mathsf{pen}}$ is defined as a finite list of roles whose traces are the standard Dolev-Yao operations.[6] The role schemas are:

$$
\begin{array}{rcll}
\mathsf{atom}_X & : & [+x] & x : X,\ X \in \{\mathsf{Name}, \mathsf{Text}, \mathsf{Data}, \mathsf{Tag}, \mathsf{Skey}, \mathsf{Akey}\} \\
\mathsf{pair} & : & [-g,\ \ -h,\ \ +(g,h)] & \\
\mathsf{sep} & : & [-(g,h),\ \ +g,\ \ +h] & \\
\mathsf{enc} & : & [-k,\ \ -m,\ \ +\{|m|\}_k] & \\
\mathsf{dec} & : & [-k^{-1},\ \ -\{|m|\}_k,\ \ +m] & \\
\mathsf{hash} & : & [-m,\ \ +\mathsf{hash}(m)]. &
\end{array}
$$

Here $x$ ranges over atoms of the indicated sort, while $g$, $h$, and $k$ range over messages. The atom roles model basic values that are generated by guessing or are initially available. The remaining roles say that the adversary may build pairs, split pairs, encrypt when an encryption key is available, decrypt when the inverse key is available, and hash any message it has received.

To analyze an honest protocol $P$ in a hostile environment, one forms the combined protocol $P \cup P_{\mathsf{pen}}$ by combining the penetrator roles and the regular roles. The operational semantics is unchanged: the adversary is just another source of role prefixes, and the bundle produced by soundness may contain both regular strands and penetrator strands.

### 4.4. Some terminology

We gather here some terminology we will use in the two following sections.

1. When $a$ and $b$ are sequences of any kind, $a \frown b$ is the result of appending them. This is the Rocq operator written ++.
2. We define $a \bowtie x = a \frown [x]$, i.e. the result of appending the one-element sequence $[x]$ to the end of $a$.
3. Analogously, we define $a \bowtie x \bowtie b = a \frown [x] \frown b$, to append $[x]$ between $a$ and $b$.
4. When $tr$ is a sequence of (ground) message transmissions and receptions, we say that $tr$ is *prefix-compatible* with $P$, written $P \rhd tr$, iff it is a prefix of some instance of a role of $P$:

$$
\begin{aligned}
P \rhd tr \quad \iff \quad & \exists r, \eta, suffix.\ r \in \mathsf{Roles}(P) \wedge \\
& \mathsf{instantiate_trace}_\eta(\mathsf{tr}(r)) = tr \frown suffix.
\end{aligned}
$$

5. If $T$ is a list of lists, then $T \downarrow e$ holds iff for some $\ell$, $e$ is a member of $\ell$ and $\ell$ is a member of $T$.

When we use $T \downarrow e$ in Section 5, $T$ will be a list of traces $\ell$, i.e. sequences of events which are ground message transmissions and receptions. Then $T \downarrow e$ says that the event $e$ occurs in some trace $\ell$ in $T$. The operational semantics does not act on parametric strands; it acts on ground traces all of which are prefix-compatible with the protocol $P$.

## 5. An operational semantics

In the last section we provided a syntax close to that of CPSA for specifying protocols, and we described its meaning. We now introduce the operational semantics of a protocol $P$.[7] A protocol $P$ determines which ground traces are compatible with that protocol, as expressed by $P \rhd tr$. Since each transition in the semantics either appends one event to an existing trace or "spawns" a new one, our operational semantics records the events in the order in which they are generated. By contrast, the bundle semantics records the causal dependencies between nodes. We will show in Section 6 that these two ways of giving a meaning to our syntax carry the same information.

### 5.1. Reachable thread pools

A *thread* is a finite ground trace, i.e., a list of directed messages. A *thread pool* is a finite list of threads.

The states of our operational semantics will be thread pools $T$ such that, for each thread $tr \in T$, $P \rhd tr$; that is, each thread has executed a sequence of events that is compatible with at least one role of the protocol.

We use $T \bowtie tr$ to describe the pool obtained by appending $tr$ as a new thread to $T$, and $T_1 \bowtie tr \bowtie T_2$ for the pool in which one relevant thread is $tr$, with surrounding threads $T_1$ and $T_2$. Either of $T_1$ and $T_2$ may be empty. Moreover, $T \downarrow e$ says that the event $e$ has occurred in some thread of the pool $T$.

Our (big-step) operational semantics uses the reachability judgment $P \vDash T$. It will mean that the thread pool $T$ can be generated from the empty pool by threads executing steps compatible with the protocol $P$. The rules are in Fig. 5, and they are mechanized by the Rocq inductive relation `reachable_pool`. Each step is constrained to advance a thread $tr$ only by an event $e$ that remains compatible with a role of the protocol $P$, as expressed in the prefix compatibility relation $P \rhd (tr \bowtie e)$.

[6] See `CPSA/PenetratorProtocol.v` in the artifact for the mechanization.
[7] See `CPSA/Semantics.v` in the artifact for the full mechanization.

$$\frac{}{P \vDash []}\ \textsc{Base} \qquad \frac{P \vDash T \qquad P \rhd [+m]}{P \vDash T \bowtie [+m]}\ \textsc{SpawnSend} \qquad \frac{P \vDash T_1 \bowtie tr \bowtie T_2 \qquad P \rhd tr \frown [+m]}{P \vDash T_1 \bowtie (tr \frown [+m]) \bowtie T_2}\ \textsc{ExtendSend}$$

$$\frac{P \vDash T \qquad T \downarrow +m \qquad P \rhd [-m]}{P \vDash T \bowtie [-m]}\ \textsc{SpawnRecv}$$

$$\frac{P \vDash T_1 \bowtie tr \bowtie T_2 \qquad T_1 \bowtie tr \bowtie T_2 \downarrow +m \qquad P \rhd tr \frown [-m]}{P \vDash T_1 \bowtie (tr \frown [-m]) \bowtie T_2}\ \textsc{ExtendRecv}$$

**Fig. 5.** The rules defining the reachability relation $P \vDash T$, parameterized by the trace compatibility relation $P \rhd tr$.

The BASE rule stipulates that the empty pool is an initial state. Of the remaining rules, two spawn new threads and two extend threads that are already present. Moreover, in each category one handles transmission events, while the other handles reception events. The send rules SPAWNSEND and EXTENDSEND require only prefix compatibility, while the receive rules SPAWNRECV and EXTENDRECV also require that a transmission of the message to be received has occurred somewhere in the current thread pool. This premise of each receive rule is merely existential; the resulting thread pool does not retain any information about which transmission was the witness justifying the reception. Thus, the causality information is not retained in the thread pool, which holds only the information needed to determine the future of the computation.

**Example 6.** Let $m = \{|Q, y, n|\}_{\mathsf{pk}(A)}$. Consider an execution of the yes-or-no protocol in which the answer is yes. Besides the answerer roles shown above, suppose the protocol contains the corresponding questioner behavior, with branches compatible with the traces $[+m, -y]$ and $[+m, -n]$. One possible reachable pool sequence is:

$$\begin{array}{lcll} T_0 & = & [] & \\ T_1 & = & [+m] & \textsc{SpawnSend} \\ T_2 & = & [+m] \bowtie [-m] & \textsc{SpawnRecv} \\ T_3 & = & [+m] \bowtie [-m, +y] & \textsc{ExtendSend} \\ T_4 & = & [+m, -y] \bowtie [-m, +y] & \textsc{ExtendRecv}. \end{array}$$

The first step starts the questioner thread by sending the encrypted question. The second step starts an answerer thread; its receive is enabled because $+m$ already occurs in $T_1$. The third step extends the answerer thread with the yes answer $+y$, using the role whose trace has prefix $[-m, +y]$. The fourth step extends the questioner thread with the matching receive $-y$; this receive is enabled because $+y$ already occurs in $T_3$. This gives one concrete yes-answer execution; the role-choice issue is separated below.

### 5.2. No commitment to roles

The judgment above is deliberately not role-committed since a running thread does not store a role name, a valuation, or a program counter. Each transition is permitted only if the extended trace is compatible with *some* instantiated role, but the satisfying role and valuation are not recorded as part of the state.

This design is important for protocols whose behavior is naturally represented by roles with a common prefix. In the yes-or-no protocol, for instance, the two possible answers are written with a choice construct, but this notation is elaborated into two ordinary roles that share the prefix $[-m]$ and then differ on the next event.

**Example 7** (Choice). Return to the pool $T_2 = [+m] \bowtie [-m]$ from the execution above. The second thread has performed only the common answerer prefix, so the state does not record whether it is an instance of the yes role or of the no role. It may be extended along the yes branch, giving the $T_3$ and $T_4$ shown above, but the same $T_2$ may also be extended along the no branch:

$$\begin{array}{lcll} T_3' & = & [+m] \bowtie [-m, +n] & \textsc{ExtendSend} \\ T_4' & = & [+m, -n] \bowtie [-m, +n] & \textsc{ExtendRecv}. \end{array}$$

The commitment happens only at the distinguishing EXTENDSEND step, when the answerer thread is extended with either $+y$ or $+n$. Thus choice does not require a special operational transition: after desugaring, choice is exactly prefix compatibility with one of the expanded roles.

### 5.3. Origination assumptions

Although roles in the syntax record the CPSA fields nonorig and uniqorig, these declarations have no direct impact on the transition semantics above. The core reachability judgment uses only role traces: it describes which pools of regular events can be generated by send and receive steps. Origination declarations are instead assumptions used in security proofs. Even when they appear in the protocol specification, they are treated as extra constraints on the behaviors that could be counterexamples to security goals. They are not additional operational rules. In CPSA, such assumptions are commonly used in skeletons, although declarations may also appear in roles and then be inherited by skeleton instances [1].

## 6. Semantic correctness

We now state the semantic correspondence theorems relating the operational semantics of Section 5 to the bundle semantics of Section 3. In one direction, every reachable pool gives rise to an inductive bundle representing the events in the pool. In the other direction, every protocol-compatible inductive bundle can be generated by the operational semantics. Section 7 then refines this correspondence computationally, adding the witnesses needed to make the constructions executable in Rocq.

### 6.1. Representing thread pools

The correspondence uses an auxiliary representation relation. A pool thread contains only the events that have already occurred. To connect such a finite prefix with a strand-space node, the Rocq development also records the complete instantiated strand trace that the thread is following. We write $\mathsf{full}(i)$ for the complete trace associated with pool thread $i$, and $\mathsf{ps}(i)$ for the strand assigned to that thread. The prefix condition is

$$\mathsf{Ext}(T, \mathsf{full}) \iff \forall i, th.\ T[i] = th \Rightarrow \exists u.\ \mathsf{full}(i) = th \frown u.$$

Thus every current thread is a prefix of its full instantiated trace. The representation relation[8] then says that the nodes of the bundle are exactly the positions already reached by the pool:

$$\begin{aligned}\mathsf{Rep}(B, T, \mathsf{full}, \mathsf{ps}) \iff \forall n.\ n \in \mathsf{nodes}(B) \iff \exists i, th, u.\ & T[i] = th \wedge \\ & \mathsf{strand}(n) = \mathsf{ps}(i) \wedge \\ & \mathsf{tr}(\mathsf{ps}(i)) = \mathsf{full}(i) \wedge \\ & \mathsf{full}(i) = th \frown u \wedge \\ & \mathsf{idx}(n) < |th|.\end{aligned}$$

The node $n$ is present in the bundle exactly when its strand is the strand assigned to some pool thread and its index lies inside the executed prefix of that thread.

**Example 8.** Consider again the labeled Concurrent Yahalom bundle in Fig. 3. Write $e_X$ for the event at node $X$. The inductive construction displayed in Section 3.1 can be read as the following operational order:

$$F_0,\ D_0,\ S_0,\ S_1,\ S_2,\ S_3,\ F_1,\ F_2,\ D_1,\ D_2,\ F_3.$$

After these events, with threads listed in the order in which they were first spawned, the pool is

$$\begin{array}{lll} 0 & \mathrm{Fst} & [e_{F_0}, e_{F_1}, e_{F_2}, e_{F_3}] \\ 1 & \mathrm{Snd} & [e_{D_0}, e_{D_1}, e_{D_2}] \\ 2 & \mathrm{Srv} & [e_{S_0}, e_{S_1}, e_{S_2}, e_{S_3}]. \end{array}$$

Here $\mathsf{ps}(0)$ is the first-participant strand, $\mathsf{ps}(1)$ is the second-participant strand, and $\mathsf{ps}(2)$ is the server strand. The function $\mathsf{full}$ records the complete instantiated traces for those strands; if a role has further events not shown in the bundle, those events appear in the suffix $u$ from the prefix condition above.

The pool itself records only the local histories. It does not list the communication edges $F_0 \to S_0$, $D_0 \to S_1$, $S_2 \to F_1$, $S_3 \to D_1$, and $D_2 \to F_3$. These edges are recovered by the soundness construction: every receive step is enabled by an earlier matching send. As discussed in Section 7, the computational refinement records the particular source occurrence for each receive, so the corresponding bundle can be computed rather than merely shown to exist.

### 6.2. Soundness

The first correctness theorem says that the operational semantics does not generate spurious executions.

**Theorem 4** (Soundness of the operational semantics). *If $P \vDash T$, then there exist $B$, $\mathsf{full}$, and $\mathsf{ps}$ such that*

$$\mathsf{Ext}(T, \mathsf{full}), \qquad \mathsf{IndBundle}(B), \qquad \mathsf{Rep}(B, T, \mathsf{full}, \mathsf{ps}).$$

*Consequently, $B$ is a bundle.*

In StrandsRocq this result is the theorem `bundle_of_reachable`; the more precise induction principle used in its proof is `bundle_of_reachable_with_trace`.

**Proof.** Fix a function $\mathsf{full}$ extending the final pool $T$; such a function always exists. We prove the more precise result `bundle_of_reachable_with_trace` by induction on the derivation of $P \vDash T$. The induction invariant provides an inductive bundle whose nodes are exactly the executed positions of the current pool, using the canonical strand assignment $\mathsf{ps}(i) = (i, \mathsf{full}(i))$. The empty derivation is represented by the empty bundle.

[8] This is the Rocq predicate `represents_pool`. Given a node $n = (s, i)$, $\mathsf{strand}(n) = s$ and $\mathsf{idx}(n) = i$ are its projections.

Consider first a send step. Spawning a thread adds the node $(\mathsf{ps}(|T|), 0)$. The extension property identifies its event with the event appended by the operational rule; the new thread index makes the node fresh. Thus SEND0 applies. To extend a selected thread $th$, add the node $(\mathsf{ps}(i), |th|)$. It is fresh because the induction invariant represents exactly the indices strictly below $|th|$. If $th$ is non-empty, its last represented node is the immediate strand predecessor, so SEND applies; if $th$ is empty, the new node has index zero and SEND0 applies. In either case the standard list decomposition around thread $i$ shows directly that the enlarged bundle represents the updated pool and that full still extends it.

For a receive step, its operational premise states that a matching transmission has already occurred. By the induction invariant, the occurrence has a positive source node $p$ in the old bundle. We add a fresh negative node $n$ together with the communication edge $p \to n$. A spawned reception has index zero and is added by RECV0. An extension of a non-empty thread also adds the strand edge from its last node and uses RECV; as in the send case, extension of an empty thread uses RECV0. The same representation argument proves the invariant for the new pool.

These cases exhaust the operational rules, and in each case yield $\mathsf{IndBundle}(B)$ together with Ext and Rep. Applying Theorem 1 shows that $B$ is an ordinary bundle. □

### 6.3. Completeness

The converse theorem says that a bundle can be scheduled as an operational history, provided it is compatible with the protocol. For a node $n$, let $\mathsf{traceAt}(n)$ be the prefix of the trace on $\mathsf{strand}(n)$ up to and including $n$. Protocol compatibility is the condition

$$\mathsf{ofProtocol}(B, P) \iff \forall n \in \mathsf{nodes}(B).\ P \rhd \mathsf{traceAt}(n).$$

Thus every node prefix appearing in the bundle is a prefix of some instantiated role of $P$.

The Rocq theorem also uses a well-formedness condition on protocols. The distinguished message $\tau$ is present in the underlying strand-space interface as a default value for out-of-bounds lookups, but it is not intended to be a protocol message. The predicate `wf_protocol_no_mesg_origination` ensures that regular roles do not originate such unconstrained message values.

The Rocq lemma `penetrator_protocol_wf_no_mesg_origination` proves that $P_{\mathsf{pen}}$ satisfies this well-formedness condition. Since this well-formedness condition is checked role by role, it is preserved when these roles are appended to any well-formed regular protocol.

In the theorem below we present this as the well-formedness assumption $\mathsf{wfProtocol}(P)$.

**Theorem 5** (Completeness of the operational semantics)**.** *Assume*

$$\mathsf{IndBundle}(B), \qquad \mathsf{ofProtocol}(B, P), \qquad \mathsf{wfProtocol}(P).$$

*Then there exist* $T$, full, *and* ps *such that*

$$P \vDash T, \qquad \mathsf{Ext}(T, \mathsf{full}), \qquad \mathsf{Rep}(B, T, \mathsf{full}, \mathsf{ps}).$$

This is the theorem `reachable_of_wf_bundle` in StrandsRocq. The development also proves a lower-level variant, `reachable_of_bundle`, in which $\mathsf{wfProtocol}(P)$ is replaced by an explicit $\mathsf{tauNonOrig}(B)$ side condition.

**Proof.** The well-formedness hypothesis implies the lower-level condition $\mathsf{tauNonOrig}(B)$ used in the Rocq proof: the distinguished default message $\tau$ does not originate in $B$. We prove the result under this condition by induction on the derivation of $\mathsf{IndBundle}(B)$. Both ofProtocol and tauNonOrig restrict to the smaller bundle in the premise of each constructor. The induction hypothesis therefore gives a reachable pool $T$, complete traces, and a strand assignment representing that smaller bundle. The empty constructor is witnessed by the empty pool.

For SEND0, the new positive node has index zero. The tauNonOrig condition, together with the inductive-bundle invariants, ensures that this index denotes an actual event of its strand rather than the out-of-bounds default. Its trace prefix is therefore a singleton send event. The ofProtocol hypothesis says that this singleton is a prefix of an instantiated role of $P$, so a SPAWNSEND step is enabled. We append the new singleton thread to $T$, assign it the strand of the new node, and take that strand's trace as its complete trace. The resulting pool represents the enlarged bundle.

For SEND, let $p \Rightarrow n$ be the new strand edge. The representation of the smaller bundle places $p$ in some thread $th$. That thread ends exactly at $p$: if it extended any farther, the fresh successor $n$ would already be represented, while it cannot end earlier because it represents $p$. Consequently $|th| = \mathsf{idx}(n)$, the next event in its complete trace is the event at $n$, and $\mathsf{traceAt}(n) = th \frown [e_n]$. Protocol compatibility supplies the required role-prefix premise, so EXTENDSEND appends $e_n$ to that thread. The complete trace and strand assignment need not change, and the representation relation is preserved.

The receive cases use the same construction plus their communication edge. In RECV0, the source node already belongs to the smaller bundle; its representation shows that the matching send event has occurred in $T$. Together with the singleton prefix supplied by ofProtocol, this enables SPAWNRECV. In RECV, the strand predecessor again identifies a thread ending immediately before the new node, while the represented communication source establishes occurrence of the matching send. Thus EXTENDRECV is enabled and preserves both Ext and Rep. These four cases construct the required witnesses under $\mathsf{tauNonOrig}(B)$; deriving that condition from $\mathsf{wfProtocol}(P)$ yields the stated theorem. □

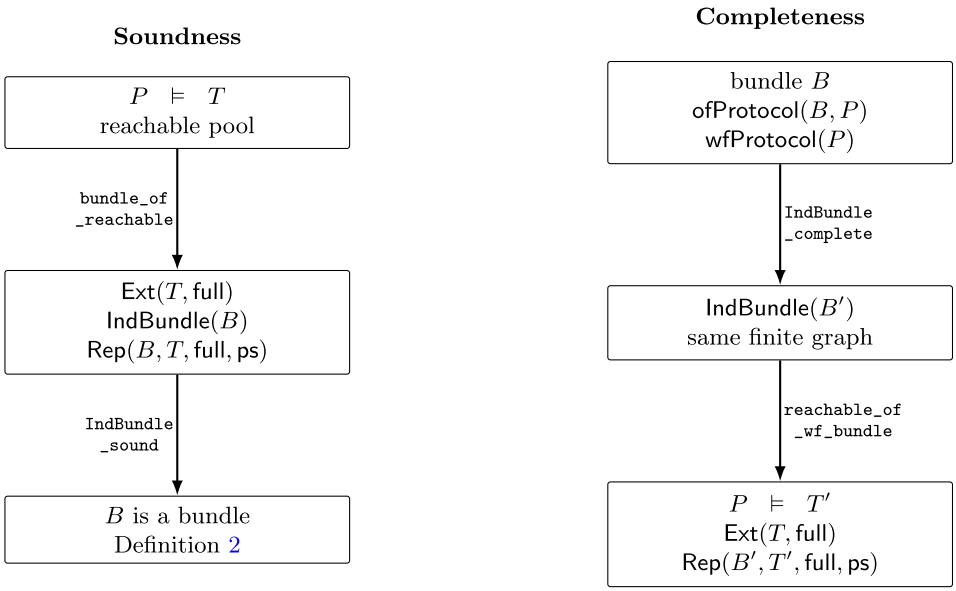



**Fig. 6.** Semantic correspondence between reachable pools and bundles.

### 6.4. *Putting the results together*

Soundness and completeness relate the two semantic views. Starting from an operational history, soundness constructs an inductive bundle, and the soundness theorem for inductive bundles turns it into a standard bundle. Starting from a standard bundle, the completeness theorem of Section 3.4 first reorders its finite node and edge lists into an inductive presentation. If the bundle is protocol-compatible, operational completeness then produces a reachable pool whose executed prefixes are exactly the nodes of that bundle.

Fig. 6 summarizes the two directions. The left column starts from the operational semantics and forgets the accidental scheduling order, obtaining a bundle. The right column starts from a protocol-compatible bundle and constructs an operational schedule for it.

Thus, the operational semantics is neither too permissive nor too restrictive with respect to strand-space executions. It generates precisely linearizations of protocol-compatible bundles, while the bundle representation forgets the accidental order between causally independent events.

## 7. Computational labeled semantics

The rules of Fig. 5 live in `Prop`, Rocq's universe of propositions, as is customary for mathematical facts. However, to prevent impredicativity from yielding contradictions like Girard's paradox [19,20] Rocq restricts *large elimination* of proof terms of these relations, making it impossible, for instance, to pattern-match on a derivation in the operational semantics and compute with the extracted information.[9] In our case, this concerns the *receive* rules. They each have a premise that there exists a matching send that occurred somewhere in the pool. A proof of this premise establishes that some matching send exists, but no witness for the thread and position of that occurrence is recoverable for computation.

To make this information available within Rocq's logic, and thereby enable computations on these objects—which typically require pattern-matching—we switch from notions in `Prop` to those within `Type`. `Type` is restricted to predicative forms of definition, and is thus a less opaque component of Rocq's universe.

### 7.1. *Unlabeled and labeled computable semantics*

We carry this out in two forms, of which we will describe only the second in detail here. In both versions, we use the same structure for the transition rules. However, we replace each occurrence of $T \downarrow e$ in a *receive* rule with a predicate $T \downarrow_{i,j} e$ that exposes the thread index $i$ and position $j$ of the transmission event $e$ that will justify the reception. Receive steps therefore acquire information witnessing the source of a communication edge being added.

In the first version[10] we define a bundle-producing judgment, $P \models^{\top} T, B$ that builds a thread pool $T$ and a bundle $B$ in lockstep using a protocol $P$. Steps add intra-strand edges where needed, and they add a communication edge from the indexed source event

[9] Except in trivial cases; see https://rocq-prover.org/doc/V9.2.0/refman/language/core/inductive.html#the-match-with-end-construction for details.

[10] See the file `CPSA/ComputableSemantics.v`.

$$\frac{P \triangleright [+m]}{P \Vdash^{\top} T \xrightarrow{\langle (|T|,0);+m;\bot\rangle} T \bowtie [+m]} \text{ L-SpawnSend}$$

$$\frac{P \triangleright tr \frown [+m]}{P \Vdash^{\top} T_1 \bowtie tr \bowtie T_2 \xrightarrow{\langle (|T_1|,|tr|);+m;\bot\rangle} T_1 \bowtie (tr \frown [+m]) \bowtie T_2} \text{ L-ExtendSend}$$

$$\frac{T \downarrow_{i,j} +m \qquad P \triangleright [-m]}{P \Vdash^{\top} T \xrightarrow{\langle (|T|,0);-m;(i,j)\rangle} T \bowtie [-m]} \text{ L-SpawnRecv}$$

$$\frac{T_1 \bowtie tr \bowtie T_2 \downarrow_{i,j} +m \qquad P \triangleright tr \frown [-m]}{P \Vdash^{\top} T_1 \bowtie tr \bowtie T_2 \xrightarrow{\langle (|T_1|,|tr|);-m;(i,j)\rangle} T_1 \bowtie (tr \frown [-m]) \bowtie T_2} \text{ L-ExtendRecv}$$

**Fig. 7.** The rules defining the labelled semantics $P \Vdash^{\top} T \xrightarrow{\ell} T'$.

to the new event when that is a reception. Using this notion, we prove that the reachable thread pools $T$ are precisely those that represent a bundle $B$. In examples we can also compute a bundle from a thread pool for a simple unilateral protocol.

### 7.2. The labeled semantics

The second version is a labeled transition system[11] which embeds the witness information in the sequence of transition labels forming the traces as the LTS executes. A label has the form $\ell = \langle d; e; s\rangle$, where:

$d = (i', j')$ indicates that the new event lies at position $j'$ on thread $i'$ in the thread pool;

$e$ is the new event being added; and

$s$ is $\bot$ if $e$ is a transmission and $s$ is $(i, j)$ if $e$ is a reception and $T \downarrow_{i,j} e_0$ is the matching transmission justifying adding $e$ at position $d$.

For a single labeled step $\ell$, and a finite labeled execution whose list of labels is $L$, we respectively write:

$$P \Vdash^{\top} T \xrightarrow{\ell} T' \quad \text{and} \quad P \Vdash^{\top} T \xrightarrow{L}{}^{*} T'$$

The step rules are analogous to the unlabeled semantics, but the receive rules expose now the indexed source occurrence $s$.

The notation $|T|$ denotes the number of threads in the current pool, and $|tr|$ denotes the length of the selected thread. Thus the destination coordinate is the thread and position of the event just created. For sends there is no communication source. For receives, the source coordinate is the indexed positive occurrence that enabled the step.

The labeled system of Fig. 7 is not a substantially different semantics. It simply makes available information that is already encoded in the computational derivation justifying judgments $P \vDash T$ in the semantics of Fig. 5. When a *receive* constructor is applied using those rules, there is a derivation of $T \downarrow +m$, and since this is existential (*some* thread $i$ sends $m$ at *some* position $j$), this derivation contains the witness $(i, j)$ embedded within it. The labeled presentation moves that same $(i, j)$ into the transition label. Consequently, labeled executions and the derivations that establish computational reachability judgments determine one another:

$$P \Vdash^{\top} [] \xrightarrow{L}{}^{*} T \qquad \text{corresponds to} \qquad P \vDash T.$$

Forgetting the labels recovers the computational pool derivation. Conversely, reading the computational derivation from left to right produces the label list $L$; in a receive case, the indexed witness $(i, j)$ becomes the source component of the label.[12]

However, the labeled semantics is very desirable for our next goal, namely proving that *every* total ordering of events compatible with a bundle's causal ordering is a possible run yielding a thread pool that represents it. The label sequences provide the orderings of the events. First, we provide an example.

**Example 9** (Labeled example)**.** For the yes-answer execution above, the labeled LTS records not only the pool sequence $T_0, \ldots, T_4$, but also the nodes created at each step and the source used by each receive:

$$\begin{aligned}
\ell_1 &= \langle (0,0); +m; \bot\rangle \\
\ell_2 &= \langle (1,0); -m; (0,0)\rangle \\
\ell_3 &= \langle (1,1); +y; \bot\rangle \\
\ell_4 &= \langle (0,1); -y; (1,1)\rangle.
\end{aligned}$$

[11] See the file `CPSA/LabelledSemantics.v`

[12] The slightly more general lemmas supporting this (informal) claim, together with their proofs, can be found in `CPSA/LabelledSemantics.v` files, called `reachable_poolT_has_labelsT` and `bundle_has_labelled_executionT`.

Thus

$$P \Vdash^{\top} [] \xrightarrow{[\ell_1,\ell_2,\ell_3,\ell_4]}{}^{*} T_4.$$

The second label says that the answerer's receive of $m$ is justified by the questioner's send at pool node $(0,0)$. The fourth label says that the questioner's receive of $y$ is justified by the answerer's send at node $(1,1)$. Once a strand assignment maps thread indices to bundle strands, these source and destination coordinates determine the corresponding interstrand edges.

### 7.3. Labeled runs cover all compatible enumerations

We start with a few definitions and initial results.

*Preliminaries.* Consider a type $A$ and a list $L$ of elements of type $A$. A function $f : \mathbb{N} \to A$ from natural numbers is *injective below* $k \in \mathbb{N}$ *into* $L$ iff for all $i < j < k$, $f(i) \neq f(j)$, and for all $i < k$, $f(i)$ is in the list $L$. It is *surjective below* $k$ *onto* $L$ iff for all $a$ in $L$, there is an $i < k$ such that $f(i) = a$. The function $f : \mathbb{N} \to A$ *enumerates* $L$ *below* $k$ if it is injective below $k$ into $L$ and surjective below $k$ onto $L$.

These notions have a familiar relation with cardinality: Injectivity ensures that the length of $L$ is at least $k$, i.e. $|L| \geq k$. Surjectivity ensures that $|L| \leq k$ if $L$ has no duplicate occurrences of any $a \in A$. Thus, if $L$ represents a set and lacks duplicates, and $f$ enumerates $L$ below $k$, then the cardinality of the set $L$ represents is $k$. Moreover, if $f$ enumerates $L$ below $|L|$, then $L$ lacks duplicates.

Another result we use below concerns omitting a value from an enumeration. We define $omit(f, i)$ to be the function $\lambda j.$ if $j < i$ then $f(j)$ else $f(j+1)$. Also, the *remove* function, applied to a value $a$ and a list $L$, yields the list containing all of the non-$a$ members of $L$ in the same order. Now, suppose $i < |L|$, $f(i) = a$, and $L' = remove(a, L)$. If $f$ enumerates $L$ below $|L|$, then $omit(f, i)$ enumerates $L'$ below $|L'|$.

Thus, given an enumeration, $omit$ gives us enumerations of sublists.

Suppose that $R$ is a strict partial order on $A$, meaning a transitive, asymmetric relation $R \subseteq A \times A$.[13] A function $f$ is *compatible with* $R$ *up to* $k$ iff for all $i < k$ and $j < k$, $R(f(i), f(j))$ implies $i < j$. Given $R$, a *compatible enumeration of* $L$ up to $k$ is an enumeration of $L$ below $|L|$ that is compatible with $R$ up to $k$. Compatibility is also preserved using $omit$ under removing elements from $L$.

*The main result.* Given a protocol $P$ and a bundle $\mathcal{B}$ of that protocol with $k$ nodes, we would like to show that, given any enumeration $f$ of its nodes in an order compatible with $\prec_{\mathcal{B}}$, there will be a label sequence $\langle \ell_0, \ldots, \ell_{k-1} \rangle$ where the destination event $d_i$ of each label $\ell_i$ represents the node $f(i)$. In more detail:

**Theorem 6** (Realization of compatible enumerations)**.** *Let $\mathcal{B}$ be a bundle of a well-formed protocol $P$, with $|nodes(\mathcal{B})| = k$. Let $\alpha$ be an enumeration of $nodes(\mathcal{B})$ up to $k$ that is compatible with $\prec_{\mathcal{B}}$. There exist a pool $T$, complete traces* full*, a strand assignment* ps : $|T| \to \Sigma$*, and a label sequence $L = \langle \ell_0, \ldots, \ell_{k-1} \rangle$ such that (i)*

$$P \Vdash^{\top} [] \xrightarrow{L}{}^{*} T, \qquad \mathsf{Ext}(T, \mathsf{full}), \qquad \mathsf{Rep}(B, T, \mathsf{full}, \mathsf{ps});$$

*moreover, (ii) for every $i < k$, if the destination of the $i$th label is $d_i = (j, h)$, then* $(\mathsf{ps}(j), h) = \alpha(i)$.

This result is mechanized in Rocq.[14]

**Proof.** We use the sink-removal induction principle for bundles, Cor. 1. The empty bundle is represented by the empty pool and the empty label sequence. For a nonempty bundle, let $k = |\mathcal{N}| - 1$ and $m = \alpha(k)$. The compatibility assumption makes $m$ a sink: if $m \prec_B n$, writing $n = \alpha(j)$ would give $k < j$, which is impossible. Remove $m$ and its incoming edges. Sink removal preserves the bundle conditions, while the restriction of $\alpha$ to its first $k$ positions is a compatible enumeration of the remaining nodes. Moreover, both protocol compatibility and the lower-level tauNonOrig condition implied by wfProtocol($P$) restrict to this smaller bundle. The induction hypothesis therefore supplies a labeled execution that represents it and realizes this restricted enumeration.

It remains to append one transition whose destination is $m$. There are four cases, according to whether $m$ is a send or receive node and whether it is initial on its strand. An initial send enables SpawnSend; a noninitial send has its strand predecessor in the smaller bundle, whose representation identifies the thread extended by ExtendSend. For an initial receive, the source of its communication edge is already represented, so the matching send has occurred and SpawnRecv is enabled. For a noninitial receive, the smaller bundle represents both that communication source and the strand predecessor, enabling ExtendRecv. In every case, ofProtocol($B$, $P$) supplies the required role prefix, and tauNonOrig($B$) ensures that the node denotes an actual trace event rather than the out-of-bounds default.

The selected transition appends a label whose destination, under the strand assignment, is precisely $m$. It also preserves Ext and extends Rep from the smaller bundle to $B$. Appending this label to the sequence obtained by induction realizes $\alpha$ in full, completing the proof. □

Consequently, the bundle semantics gives a data structure that summarizes executions modulo permutation of independent steps.

[13] Asymmetric in the sense that not both $R(a, a')$ and $R(a', a)$ for any $a, a'$; in particular, not $R(a, a)$. The material on enumerations and compatibility is in `Common/Enumerate.v`

[14] The theorem `compatible_enumeration_has_labelled_execution` is in `CPSA/EnumerationSemantics.v`.

```
defrole "init"
  vars [ [a; b; s] is Name; [n_a; n_b] is Text; [k] is Skey ]
  trace [ send cat{a; b; n_a};
    recv {|a, b, k, n_a, n_b|}_ltk(a,s);
    send {|tag_1, a, b, n_a, n_b|}_k;
    recv {|tag_2, a, b, n_a, n_b|}_k ]
  nonorig []
  uniqorig [];
```

```
defrole "resp"
  vars [ [a; b; s] is Name; [n_a; n_b] is Text; [k] is Skey ]
  trace [ send cat{a; b; n_b};
    recv {|a, b, k, n_a, n_b|}_ltk(b,s);
    send {|tag_2, a, b, n_a, n_b|}_k;
    recv {|tag_1, a, b, n_a, n_b|}_k ]
  nonorig []
  uniqorig [];
```

```
defrole "serv"
  vars [ [a; b; s] is Name; [n_a; n_b] is Text; [k] is Skey ]
  trace [ recv cat{a; b; n_a};
    recv cat{a; b; n_b};
    send {|a, b, k, n_a, n_b|}_ltk(a,s);
    send {|a, b, k, n_a, n_b|}_ltk(b,s) ]
  nonorig []
  uniqorig [k].
```

**Fig. 8.** The Concurrent Yahalom protocol written in CPSA-like syntax.

## 8. Worked example: concurrent Yahalom

We conclude the technical development by following the results through the Concurrent Yahalom execution of Fig. 3. The point of the example is not to analyze Yahalom again, but to show how the objects introduced in the previous sections fit together for a protocol with several strands and several independent actions.

### 8.1. Ground run

The code in Fig. 8 is a CPSA-style Rocq description of the Concurrent Yahalom protocol (Fig. 3), written in the same notation as Section 4.2. The roles ``init'' and ``resp'' are deliberately symmetric after their initial requests, so that the peer actions can interleave maximally.

The Rocq file `CPSA/PaperExamples.v` contains the corresponding role definitions and proves that their protocol is well-formed for the semantic results used here. The variables $a$, $b$, and $s$ have sort Name; $n_a$ and $n_b$ have sort Text; and $k$ has sort Skey. The confirmation labels $\mathsf{tag}_1$ and $\mathsf{tag}_2$ are fixed values of sort Tag, not role variables. The uniqorig $[k]$ declaration states that the server freshly originates the session key. The original CPSA example also records a side condition that the two peers are distinct; this is a CPSA-side assumption and is not used by the core reachability semantics of Section 5.

Fix principals $A$, $B$, and $S$, nonces $N_a$ and $N_b$, and a session key $K$. Let

$$\begin{aligned}
req_A &= A, B, N_a, \\
req_B &= A, B, N_b, \\
tic_A &= \{|A, B, K, N_a, N_b|\}_{\mathsf{ltk}(A,S)}, \\
tic_B &= \{|A, B, K, N_a, N_b|\}_{\mathsf{ltk}(B,S)}, \\
conf_A &= \{|\mathsf{tag}_1, A, B, N_a, N_b|\}_K, \\
conf_B &= \{|\mathsf{tag}_2, A, B, N_a, N_b|\}_K.
\end{aligned}$$

In the run shown in Fig. 3, the first participant, the server, and the second participant execute the ground traces

$$\begin{aligned}
\mathrm{Fst} &= [+req_A, -tic_A, +conf_A, -conf_B], \\
\mathrm{Srv} &= [-req_A, -req_B, +tic_A, +tic_B], \\
\mathrm{Snd} &= [+req_B, -tic_B, +conf_B].
\end{aligned}$$

The corresponding node labels are those of Fig. 3: $F_i$ for the first participant, $S_i$ for the server, and $D_i$ for the second participant.

### 8.2. From an operational history to a bundle

One reachable pool execution compatible with this run follows the same order used in Section 3.1:

$$
\begin{aligned}
T_0 &= [] \\
T_1 &= [+req_A] \\
T_2 &= [+req_A] \bowtie [+req_B] \\
T_3 &= [+req_A] \bowtie [+req_B] \bowtie [-req_A] \\
T_4 &= [+req_A] \bowtie [+req_B] \bowtie [-req_A, -req_B] \\
T_5 &= [+req_A] \bowtie [+req_B] \bowtie [-req_A, -req_B, +tic_A] \\
T_6 &= [+req_A] \bowtie [+req_B] \bowtie [-req_A, -req_B, +tic_A, +tic_B] \\
T_7 &= [+req_A, -tic_A] \bowtie [+req_B] \bowtie [-req_A, -req_B, +tic_A, +tic_B] \\
T_8 &= [+req_A, -tic_A, +conf_A] \bowtie [+req_B] \bowtie [-req_A, -req_B, +tic_A, +tic_B] \\
T_9 &= [+req_A, -tic_A, +conf_A] \bowtie [+req_B, -tic_B] \bowtie [-req_A, -req_B, +tic_A, +tic_B] \\
T_{10} &= [+req_A, -tic_A, +conf_A] \bowtie [+req_B, -tic_B, +conf_B] \bowtie [-req_A, -req_B, +tic_A, +tic_B] \\
T_{11} &= [+req_A, -tic_A, +conf_A, -conf_B] \bowtie [+req_B, -tic_B, +conf_B] \bowtie [-req_A, -req_B, +tic_A, +tic_B].
\end{aligned}
$$

The steps $T_0 \to T_1$ and $T_1 \to T_2$ use SPAWNSEND. The server thread is then started by SPAWNRECV, using the previous send of $req_A$, and extended by EXTENDRECV, using the previous send of $req_B$. The server then performs two EXTENDSEND steps. The remaining receive steps are enabled respectively by the sends of $tic_A$, $tic_B$, and $conf_B$, while the send of $conf_A$ is a local EXTENDSEND step on the first participant thread.

Soundness applies to the final reachable pool $T_{11}$. It produces a graph $B$, a full-trace assignment $\mathsf{full}$, and a strand assignment $\mathsf{ps}$ such that

$$\mathsf{IndBundle}(B) \qquad \text{and} \qquad \mathsf{Rep}(B, T_{11}, \mathsf{full}, \mathsf{ps}).$$

The representation relation identifies the nodes of $B$ with the executed positions of the three pool threads. In the Prop-level semantics, each receive step is justified by the existence of a previous matching send, and the soundness proof uses these witnesses to choose suitable communication sources. Thus, for this execution, the resulting bundle has communication edges

$$F_0 \to S_0, \quad D_0 \to S_1, \quad S_2 \to F_1, \quad S_3 \to D_1, \quad D_2 \to F_3.$$

The graph obtained from the operational history is exactly the bundle displayed in Fig. 3. Finally, the inductive soundness theorem of Section 3.2 turns $\mathsf{IndBundle}(B)$ into the ordinary bundle property of Definition 2. If we switch to the computational version discussed in Section 7, the source occurrence for each receive is recorded as data, so these communication edges can be computed directly from the execution witness rather than merely obtained inside the soundness proof.

### 8.3. From the bundle back to an operational history

Conversely, suppose we start from the bundle in Fig. 3. Since it is finite and acyclic, it has maximal nodes. Removing a maximal node that is a sink gives a smaller bundle, as discussed in Section 3.4. Repeating this operation yields a reverse construction order. Reading the removed nodes backwards gives an inductive generation order such as

$$F_0,\ D_0,\ S_0,\ S_1,\ S_2,\ S_3,\ F_1,\ F_2,\ D_1,\ D_2,\ F_3.$$

This is precisely the order used above to obtain $T_0, \ldots, T_{11}$. Other choices of sink give different linear schedules. For instance, after $S_2$ and $S_3$ have both occurred, the receptions $F_1$ and $D_1$ are independent, so they may be scheduled in either order. These different schedules represent the same partial-order bundle.

Operational completeness now uses the protocol-compatibility condition $\mathsf{ofProtocol}(B, P)$. For each node $n$ in the bundle, the prefix $\mathsf{traceAt}(n)$ must be compatible with some instantiated role of the Concurrent Yahalom protocol. This supplies the $P \rhd tr$ premise needed by the corresponding operational rule. Communication edges supply the receive-enabling send premise. Thus the sink-removal order becomes a reachable pool execution, and the final pool represents the original bundle.

### 8.4. What the example shows

The example makes explicit the two forms of information related by the theorems. The operational history records a sequential order of thread-pool transitions, but not the final partial order directly. The bundle records the causal partial order, but not a unique sequential schedule. Soundness forgets the accidental scheduling order and reconstructs the causal graph. Completeness chooses a linear schedule compatible with the causal graph and reconstructs a reachable pool. Concurrent Yahalom is useful here precisely because it contains incomparable actions: the theorem does not say that there is one canonical execution order, but that operational histories and bundles describe the same executions at different levels of observation.

## 9. Related work

*Strand spaces and their relation to other formalisms.* Strand spaces were initially introduced by Thayer et al. [4] as a symbolic framework for protocol analysis. Strand spaces have since been the subject of multiple studies and extended in several directions, including for

example [11,21–26]. In particular, Guttman and Thayer extended the strands spaces framework with skeletons, shapes, and the principle of authentication tests [5,27] enabling automated protocol analysis. This automated, search-based and complete [6,15] approach is implemented in the Cryptographic Protocol Shapes Analyzer (CPSA) tool [1] whose syntax for protocol specifications served as a basis for our Section 4. Rather than focusing on automated analysis, here we build on StrandsRocq [2], a mechanization of the original strand spaces framework in Rocq. Specifically, this work bridges the meaning of CPSA specifications with that of strand spaces' bundles via their inductive characterization and an operational semantics for CPSA syntax.

Our results on inductive bundles are similar to that of Li and Pang [18]. While their notion of inductive bundle is similar to ours, and they have an approximation to the soundness theorem we give as Theorem 1, they do not have a completeness result like Theorem 3. By contrast, Ramsdell provides an operational semantics for a single session of a role, concentrating on the stepwise construction of terms to be sent and the analysis of terms as they are received [28].

Several works have investigated the relation between strand spaces and other influential formalisms for protocol description and analysis. One of the earliest attempts in comparing strand spaces with other formalisms is due to Meadows [29], where strand spaces are compared with other early approaches, while Cervesato et al. [30] showed how to encode strand spaces in first-order linear logic, proving the encoding sound and complete. Heather [31] linked them to rank functions [32]. Cervesato et al. [22] gave a formal comparison between strand spaces and multiset rewriting, establishing a bisimulation between the two formalisms. Moving to the side of process algebras, Crazzolara and Winskel [33] map a simple process algebra (similar to the spi-calculus) to other formalisms, including strand spaces. Yang et al. [25] extend strand spaces with non-deterministic choice and give them a process-algebraic operational semantics, which is proved equivalent (up to bisimulation) to the original symbolic semantics of Maude-NPA. Unlike [25], our semantics handles choice without extending the framework, as it arises naturally from the "no commitment to roles" design choice described in Section 5.2. Halpern and Pucella [23] and Caleiro et al. [21] further connect strand spaces to multi-agent systems and distributed temporal logic, respectively. More broadly, our comparison between sequential operational histories and partially ordered bundles is related to the classical true-concurrency view of computation developed through Petri nets and event structures [34–36]. Petri-net semantics have also been used directly in security theory, notably in Busi and Gorrieri's survey of non-interference with Petri nets [37].

Unlike all these works, which establish semantic correspondences between strand spaces and other formalisms, we relate strand spaces to an operational model. In doing so, we provide an operational semantics for CPSA specifications grounded in strand spaces itself and fully mechanized in Rocq.

*Other approaches to protocol analysis.* The literature on protocol analysis is extensive, and it would be challenging to cover within a single section, for a general overview of the field we refer the reader to Barbosa et al. [38], Blanchet [39], Cortier et al. [40]. There are essentially two complementary approaches to protocol analysis, the first being the symbolic approach with its idealized model of cryptography, the second being the computational approach with its more concrete world view. While state-of-the-art tools for symbolic analysis can often automatically verify real-world protocols [1,41–43], the literature offers semi-automated, interactive symbolic techniques in which proofs are partially provided by hand. Semi-automated techniques are useful because they allow powerful inductive reasoning that automated tools do not fully support [44]. In fact, general-purpose proof assistants like Isabelle/HOL, Rocq, Lean, or F* [45] can be used to handle protocols that are not tractable by the constrained models automated tools impose to keep analysis feasible. StrandsRocq [2], the framework which we build upon and extend, is an example. It is a full mechanization of the Strand spaces framework, which provides specialized tactics to ease proving security properties about protocols while leaving the protocol designer some burden of the proof. Another influential example is DY* [44], a verification framework for symbolic protocol analysis based on F*. DY* leverages dependent types to prove protocol security, and allows for extracting protocol implementations in F*. Historically, the most influential of these approaches is due to Paulson [46]. Paulson's approach has been applied to complex and realistic protocols, demonstrating its scalability through proof automation in Isabelle (see, e.g., [47]).

## 10. Some reflections

In this paper we have studied two approaches to the semantics of protocols. The one is causal, and the other is an operational semantics. To understand the causal approach, we started with a direct formalization of the standard strand space model of execution, the bundle. It is equipped with a partial order generated by causal dependencies, each of which is either the dependence of a node on its immediate predecessor on the same strand or the dependence of a reception on an earlier node that transmits the same message. We introduce an inductive definition that generates bundles node by node, adding the necessary edges along the way. The definition is similar to the one proposed by Li and Pang [18]; like them, we prove that every inductively generated object is in fact a bundle. We also prove that every bundle has some inductive generation.

Our operational semantics is relative to a particular protocol. The protocol, which for this purpose includes both the roles executed by the compliant principals and also roles that characterize the adversary, determines the transition relation. The operational semantics acts on states that are thread pools, where each thread pool is a list of *threads*, and each thread is a list of events, i.e. message transmissions and receptions. A thread pool is insensitive to a particular assignment of strands to threads: a strand "knows how the story will end," while a thread knows only what has happened so far. Thus, it makes no commitment to a particular role or valuation for the parameters of that role. This is what enables the thread pools to reflect the choice operator of the syntax accurately. By contrast, a bundle is a graph whose vertices are nodes, and a node is a pair of a whole strand and a position possibly only partway along it. In that sense, the nodes already embed a commitment to a particular future.

Thus, the operational semantics is designed to represent the open, step-by-step nature of execution, which the bundle notion is not. A thread pool *represents* a bundle relative to a particular way to complete the events that have occurred so far so as to match the nodes in the bundle; however, the bundle ordering is not visible in the thread pool.

We proved that there is a run of the operational semantics of protocol $P$ yielding a thread pool $T$ just in case that thread pool (relative to some completion) represents a bundle of $P$. We also proved that, given a bundle and any enumeration of its nodes that is compatible with the causal ordering, there is a transition sequence that generates it following the steps of that enumeration.

An intriguing subtheme of our work connects with the Rocq logic. Since it is constructive, its proofs consist of constructions that deliver objects inhabiting the theorems. However, when we formulated the operational semantics in kind `Prop`, we no longer had computational access to the values hidden with the proofs, such as the witnesses justifying existential claims. This motivated an essentially equivalent formulation in kind `Type`, from which their computational content could be recovered. This allows a run to be generated to reach a given thread pool for instance.[15] The labeled semantics provides an interplay between computation and deduction that fulfills a vision of the pioneers of dependent type theory and the propositions-as-types interpretation [19,48].

As future work, active automata learning techniques such as those used by ALVIE [3] could provide a further bridge from the verified protocol models considered here to the behavior of concrete protocol implementations. We also envisage tighter connections between StrandsRocq and CPSA, enabling StrandsRocq to verify individual steps in the CPSA search, and enabling StrandsRocq to prove and export lemmas about a particular protocol under analysis. These lemmas would justify CPSA in pruning its search.

## CRediT authorship contribution statement

**Matteo Busi:** Writing – review & editing, Writing – original draft, Validation, Software, Methodology, Investigation, Formal analysis, Conceptualization; **Riccardo Focardi:** Writing – review & editing, Writing – original draft, Validation, Software, Methodology, Investigation, Formal analysis, Conceptualization; **Joshua Guttman:** Writing – review & editing, Writing – original draft, Validation, Software, Methodology, Investigation, Formal analysis, Conceptualization; **Flaminia Luccio:** Writing – review & editing, Writing – original draft, Validation, Software, Methodology, Investigation, Formal analysis, Conceptualization.

## Declaration of generative AI and AI-assisted technologies in the manuscript preparation process

During the preparation of this work, the authors used OpenAI's Codex and Anthropic's Claude to assist in finding parts of the Rocq proofs and in drafting early versions of some sections of the manuscript. All proofs have been checked by Rocq. The authors have thoroughly rewritten all of the AI-generated text and take full responsibility for the published article.

## Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## Acknowledgements

Thanks to Pierpaolo Degano, who years ago suggested exploring the relationship between the strand space partial orders and a linearly ordered trace semantics. We also dedicate a grateful thought to Nadia Busi, who passed away too early, and whose contributions to the semantic theory of concurrency and security remain an important part of the tradition surrounding this work.

This work is partially supported by the SEcurity and RIghts In the CyberSpace - SERICS (PE00000014 - CUP H73C2200089001) project under the National Recovery and Resilience Plan (NRRP) funded by the European Union - NextGenerationEU and Cybersecurity Certification and Assessment Tools (CCAT) project funded under grant agreement no 101225878 and supported by the European Cybersecurity Competence Centre.

Funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Cybersecurity Competence Centre. Neither the European Union nor the European Cybersecurity Competence Centre can be held responsible for them.

## Data availability

The Rocq artifact (all proofs and formalizations) is openly available on Zenodo: (https://doi.org/10.5281/zenodo.22939512).

## References

[1] M.D. Liskov, J.D. Ramsdell, J.D. Guttman, P.D. Rowe, The Cryptographic Protocol Shapes Analyzer: A Manual for CPSA 4, The MITRE Corporation, 2023. https://github.com/mitre/cpsa/blob/master/doc/cpsa4manual.pdf.

[2] M. Busi, R. Focardi, F.L. Luccio, Strands Rocq: why is a security protocol correct, mechanically?, in: 2025 IEEE 38th Computer Security Foundations Symposium (CSF), 2025, pp. 33–48. https://doi.org/10.1109/CSF64896.2025.00022

[15] as in `CPSA/PaperExamples.v`.

[3] M. Busi, R. Focardi, F.L. Luccio, Bridging the gap: automated analysis of sancus, in: 2024 IEEE 37th Computer Security Foundations Symposium (CSF), IEEE, 2024, pp. 233–248. https://doi.org/10.1109/CSF61375.2024.00023

[4] F.J. Thayer Fábrega, J.C. Herzog, J.D. Guttman, Strand spaces: proving security protocols correct, J. Comput. Secur. 7 (2–3) (1999) 191–230. https://doi.org/10.3233/JCS-1999-72-304

[5] J.D. Guttman, F.J. Thayer, Authentication tests and the structure of bundles, Theor. Comput. Sci. 283 (2) (2002) 333–380. https://doi.org/10.1016/S0304-3975(01)00139-6

[6] S.F. Doghmi, J.D. Guttman, F.J. Thayer, Completeness of the authentication tests, in: J. Biskup, J. Lopez (Eds.), European Symposium on Research in Computer Security (ESORICS), (4734) in LNCS, Springer-Verlag, 2007, pp. 106–121.

[7] J.D. Guttman, F.J. Thayer, Protocol independence through disjoint encryption, in: Computer Security Foundations Workshop, IEEE CS Press, 2000.

[8] J.D. Guttman, Cryptographic protocol composition via the authentication tests, in: L. de Alfaro (Ed.), Foundations of Software Science and Computation Structures (FOSSACS), number 5504 in LNCS, Springer, 2009, pp. 303–317.

[9] J.D. Guttman, Establishing and preserving protocol security goals, J. Comput. Secur. 22 (2) (2014) 201–267.

[10] J.D. Guttman, Authentication tests and disjoint encryption: a design method for security protocols, J. Comput. Secur. 12 (3/4) (2004) 409–433.

[11] S.L. Dyer, C.A. Femrite, J.D. Guttman, J.P. Lanson, M.D. Liskov, Cryptographically assured information flow: assured remote execution, in: IEEE Computer Security Foundations Symposium, 2025, pp. 1–16. http://web.cs.wpi.edu/~guttman/pubs/article_arex.pdf.

[12] M. Burrows, M. Abadi, R. Needham, A logic of authentication, Proc. R. Soc. Series A, 426 (1871) (1989) 233–271. Also appeared as SRC Research Report 39 and, in a shortened form, in ACM Transactions on Computer Systems 8, 1 (February 1990), 18–36.

[13] L.C. Paulson, On Two Formal Analyses of the Yahalom Protocol, Technical Report 432, Cambridge University Computer Laboratory, 1997.

[14] J. Clark, J. Jacob, A Survey of Authentication Protocol Literature: Version 1.0, 1997. University of York, Department of Computer Science, https://eprints.whiterose.ac.uk/id/eprint/72494/1/drareviewps.pdf.

[15] J.D. Guttman, Shapes: surveying crypto protocol runs, in: V. Cortier, S. Kremer (Eds.), Formal Models and Techniques for Analyzing Security Protocols, Cryptology and Information Security Series, IOS Press, 2011, pp. 222–57.

[16] P.D. Rowe, J.D. Guttman, M.D. Liskov, Measuring protocol strength with security goals, Int. J. Inf. Secur. 15 (6) (2016) 575–596. DOI 10.1007/s10207-016-0319-z, http://web.cs.wpi.edu/~guttman/pubs/ijis_measuring-security.pdf.

[17] D. Dolev, A. Yao, On the security of public-key protocols, IEEE Trans. Inf. Theory 29 (1983) 198–208.

[18] Y. Li, J. Pang, An inductive approach to strand spaces, Form. Asp. Comput. 25 (4) (2013) 465–501.

[19] P. Martin-Löf, An intuitionistic theory of types: predicative part, in: Studies in Logic and the Foundations of Mathematics, 80, Elsevier, 1975, pp. 73–118.

[20] A.J.C. Hurkens, A simplification of Girard's paradox, in: M. Dezani-Ciancaglini, G.D. Plotkin (Eds.), Typed Lambda Calculi and Applications TLCA, vol. 902 of Lecture Notes in Computer Science, Springer, 1995, pp. 266–278. https://doi.org/10.1007/BFB0014058

[21] C. Caleiro, L. Viganò, D.A. Basin, Relating strand spaces and distributed temporal logic for security protocol analysis, Log. J. IGPL 13 (6) (2005) 637–663. https://doi.org/10.1093/JIGPAL/JZI048

[22] I. Cervesato, N.A. Durgin, P. Lincoln, A comparison between strand spaces and multiset rewriting for security protocol analysis, J. Comput. Secur. 13 (2) (2005) 265–316. https://doi.org/10.3233/JCS-2005-13203

[23] J.Y. Halpern, R. Pucella, On the relationship between strand spaces and multi-agent systems, ACM Trans. Inf. Syst. Secur. 6 (1) (2003) 43–70.

[24] A. Kamil, G. Lowe, Understanding abstractions of secure channels, in: P. Degano, S. Etalle, J.D. Guttman (Eds.), Formal Aspects of Security and Trust - 7th Int. Workshop. Revised Selected Papers, vol. 6561 of Lecture Notes in Computer Science, Springer, 2010, pp. 50–64. https://doi.org/10.1007/978-3-642-19751-2_4

[25] F. Yang, S. Escobar, C. Meadows, J. Meseguer, S. Santiago, Strand spaces with choice via a process algebra semantics, in: P. López-García, J.P. Gallagher, R. Giacobazzi (Eds.), Analysis, Verification and Transformation for Declarative Programming and Intelligent Systems - Essays Dedicated to Manuel Hermenegildo on the Occasion of His 60th Birthday, LNCS, Springer, 2023, pp. 307–350. https://doi.org/10.1007/978-3-031-31476-6_17

[26] A.T. Sherman, J.J.R. Romano, E. Zieglar, E. Golaszewski, J.D. Fuchs, W.E. Byrd, Analysis of the security design, engineering, and implementation of the secureDNA system, in: Network and Distributed System Security (NDSS), The Internet Society, 2026. https://arxiv.org/pdf/2512.09233.

[27] S.F. Doghmi, J.D. Guttman, F.J. Thayer, Searching for shapes in cryptographic protocols, in: Tools and Algorithms for the Construction and Analysis of Systems, vol. 4424 of Lecture Notes in Computer Science, Springer, 2007, pp. 523–538. https://doi.org/10.1007/978-3-540-71209-1_41

[28] J. D. Ramsdell, Cryptographic protocol analysis and compilation using CPSA and roletran, in: D. Dougherty, J. Meseguer, S. A. Mödersheim, P. D. Rowe (Eds.), Protocols, Strands, and Logic - Essays Dedicated to Joshua Guttman on the Occasion of his 66.66th Birthday, Vol. 13066 of Lecture Notes in Computer Science, Springer, 2021, pp. 355–369. doi:10.1007/978-3-030-91631-2_20.

[29] C. Meadows, Invariant generation techniques in cryptographic protocol analysis, in: Proceedings of the 13th IEEE Computer Security Foundations Workshop, CSFW '00, Cambridge, England, UK, July 3–5, 2000, IEEE Computer Society, 2000, pp. 159–167. https://doi.org/10.1109/CSFW.2000.856934

[30] I. Cervesato, N. Durgin, M. Kanovich, A. Scedrov, Interpreting strands in linear logic, 2000, (https://apps.dtic.mil/sti/pdfs/ADA465155.pdf).

[31] J. Heather, Strand spaces and rank functions: more than distant cousins, in: 15th IEEE Computer Security Foundations Workshop (CSFW-15 2002), 24–26 June 2002, Cape Breton, Nova Scotia, Canada, IEEE Computer Society, 2002, p. 104. https://doi.org/10.1109/CSFW.2002.1021810

[32] S.A. Schneider, Verifying authentication protocols with CSP, in: 10th Computer Security Foundations Workshop (CSFW '97), June 10–12, 1997, Rockport, Massachusetts, USA, IEEE Computer Society, 1997, pp. 3–17. https://doi.org/10.1109/CSFW.1997.596775

[33] F. Crazzolara, G. Winskel, Events in security protocols, in: Proceedings, 8th ACM Conference on Computer and Communications Security, ACM Press, Philadelphia, 2001, pp. 96–105.

[34] M. Nielsen, G.D. Plotkin, G. Winskel, Petri nets, event structures and domains, part I, Theor. Comput. Sci. 13 (1) (1981) 85–108. https://doi.org/10.1016/0304-3975(81)90112-2

[35] G. Winskel, Event structures, in: Advances in Petri Nets 1986, Part II, vol. 255 of Lecture Notes in Computer Science, Springer, 1987, pp. 325–392. https://doi.org/10.1007/3-540-17906-2_31

[36] G. Winskel, M. Nielsen, Models for concurrency, in: Handbook of Logic in Computer Science, 4, Oxford University Press, 1995, pp. 1–148.

[37] N. Busi, R. Gorrieri, A survey on non-interference with petri nets, in: J. Desel, W. Reisig, G. Rozenberg (Eds.), Lectures on Concurrency and Petri Nets, vol. 3098 of Lecture Notes in Computer Science, Springer, 2003, pp. 328–344. https://doi.org/10.1007/978-3-540-27755-2_8

[38] M. Barbosa, G. Barthe, K. Bhargavan, B. Blanchet, C. Cremers, K. Liao, B. Parno, SoK: computer-aided cryptography, in: 42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24–27 May 2021, IEEE, 2021, pp. 777–795. https://doi.org/10.1109/SP40001.2021.00008

[39] B. Blanchet, Security protocol verification: symbolic and computational models, in: P. Degano, J.D. Guttman (Eds.), Principles of Security and Trust, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 3–29.

[40] V. Cortier, S. Kremer, B. Warinschi, A survey of symbolic methods in computational analysis of cryptographic systems, J. Autom. Reason. 46 (3–4) (2011) 225–259. https://doi.org/10.1007/s10817-010-9187-9

[41] B. Blanchet, B. Smyth, V. Cheval, M. Sylvestre, ProVerif 2.02pl1: Automatic Cryptographic Protocol Verifier, User Manual and Tutorial, 2020, (Available at https://prosecco.gforge.inria.fr/personal/bblanche/proverif/manual.pdf).

[42] S. Escobar, C. Meadows, J. Meseguer, Maude-NPA: Cryptographic Protocol Analysis Modulo Equational Properties, Springer, Berlin, Heidelberg, 2009, pp. 1–50. https://doi.org/10.1007/978-3-642-03829-7_1

[43] S. Meier, B. Schmidt, C. Cremers, D. Basin, The TAMARIN prover for the symbolic analysis of security protocols, in: 25th International Conference on Computer Aided Verification, 2013, pp. 696–701. https://doi.org/10.1007/978-3-642-39799-8_48

[44] K. Bhargavan, A. Bichhawat, Q.H. Do, P. Hosseyni, R. Küsters, G. Schmitz, T. Würtele, DY*: a modular symbolic verification framework for executable cryptographic protocol code, in: 2021 IEEE European Symposium on Security and Privacy, 2021, pp. 523–542. https://doi.org/10.1109/EuroSP51992.2021.00042

[45] N. Swamy, J. Chen, C. Fournet, P. Strub, K. Bhargavan, J. Yang, Secure distributed programming with value-dependent types, J. Funct. Program. 23 (4) (2013) 402–451. https://doi.org/10.1017/S0956796813000142

[46] L.C. Paulson, The inductive approach to verifying cryptographic protocols, J. Comput. Secur. 6 (1–2) (1998) 85–128. https://doi.org/10.3233/JCS-1998-61-205

[47] G. Bella, L.C. Paulson, Kerberos version 4: inductive analysis of the secrecy goals, in: J. Quisquater, Y. Deswarte, C. Meadows, D. Gollmann (Eds.), 5th European Symposium on Research in Computer Security, vol. 1485 of Lecture Notes in Computer Science, Springer, 1998, pp. 361–375. https://doi.org/10.1007/BFB0055875

[48] W.A. Howard, The formulae-as-types notion of construction, in: To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism, Academic Press, 1980, p. 479–491.