Causality-based Abstraction of Multiplicity in Security Protocols

Michael Backes Matteo Maffei
Saarland University, Saarbrücken, Germany
{backes,maffei}@cs.uni-sb.de

Agostino Cortesi
Ca' Foscari University, Venice, Italy
cortesi@dsi.unive.it

Abstract

This paper presents a novel technique for analyzing security protocols based on an abstraction of the program semantics. This technique is based on a novel structure called causal graph which captures the causality among program events within a finite graph. A core property of causal graphs is that they abstract away from the multiplicity of protocol sessions, hence constituting a concise tool for reasoning about an even infinite number of concurrent protocol sessions; deciding security only requires a traversal of the causal graph, thus yielding a decidable, and typically very efficient, approach for security protocol analysis. Additionally, causal graphs allow for dealing with different security properties such as secrecy and authenticity in a uniform manner. Both the construction of the causal graph from a given protocol description and the analysis have been fully automated and tested on several example protocols from the literature.

1 Introduction

Proofs of security protocols are known to be error-prone and, owing to the distributed-system aspects of multiple interleaved protocol runs, awkward to make for humans. In fact, vulnerabilities have accompanied the design of such protocols ever since early authentication protocols like Needham-Schroeder [15, 29], over carefully designed de-facto standards like SSL and PKCS [31, 7], up to current widely deployed products like Microsoft Passport [20]. Formal methods, and in particular language-based techniques, have proved to constitute important tools for dealing with such flaws, by helping both to securely design and to analyze security protocols. A central intricacy that these approaches have to tackle is how to concisely treat the potentially unbounded number of concurrent protocol sessions. Techniques based on state-space exploration rely on the inspection of (abstractions of) all the unbounded sequences of messages exchanged by honest principals, and although such approaches often allow for a careful detection and reproduction of successful attacks, they are inherently constrained by only taking a finite number of sessions into account, or by restricting the analysis to certain classes of protocols, or by giving up guaranteed termination. In contrast to these approaches, static analysis techniques based on type systems rely on the identification of some syntactic patterns that suffice for guaranteeing the property of interest. Furthermore, type systems enjoy guaranteed termination since they work on the syntax of the protocol. The drawback of these techniques is that for analyzing different security properties one has to identify multiple different patterns and, consequently, to develop different type systems. Furthermore, the aforementioned patterns are sometimes restrictive in that, e.g., ensuring freshness of authentication requests presupposes that every authentication session is based on a different nonce (an unguessable random string) that is used only in a single protocol session and then discarded [23, 12]. This rules out some interesting protocols relying on composing multiple authentication sessions based on the same nonce, e.g., see [26].

This paper presents a novel technique for analyzing security protocols based on abstract interpretation of program semantics [13]. In particular, this technique abstracts away from the multiplicity of protocol sessions while still preserving the causality among program events. More precisely, the unbounded number of execution traces of a protocol generated by running an unbounded number of concurrent protocol sessions is concisely abstracted into a novel structure that we call *causal graph*, i.e., a finite graph in which nodes represent process events and edges express the causality among events. Interestingly, causal graphs allow for soundly characterizing which terms abstract messages generated in the same protocol session and which ones may instead abstract messages generated in different protocol sessions; this information turns out to be crucial in order to determine the safety of protocol specifications. In contrast to state-space exploration techniques, our analysis enjoys guaranteed termination. Furthermore, deciding security requires only a traversal of the causal graph, and causal graphs turn out to be of decent size for commonly analyzed security protocols. Our work improves on existing type-based approaches in that we do not restrict the class of analyzable protocols to the ones adhering to some specific syntactic patterns; this is possible since our analysis is based on an abstraction of the program semantics. Finally, our approach allows for a uniform treatment of different security properties such as secrecy and two different variants of authenticity, namely non-injective agreement and agreement [28].

We can summarize our technique as follows. We first specify the protocol in a dialect of the spi-calculus [4], a process calculus for modelling cryptographic protocols. We then construct the corresponding causal graph, which is proven to be unique for every protocol. The causal graph then allows for analyzing the intended security properties. Since causal graphs are of finite size, the analysis is assured to terminate. Finally, the safety of the causal graph implies the safety of the protocol and, more precisely, the safety of the possibly unbounded number of protocol execution traces. As usual for static analysis and due to the undecidability of the original problem, failures in the verification may be caused by either a flaw in the protocol or a non-sufficient precision of the analysis ruling out safe protocols.

Finally, the analysis is amenable to full automation. We have implemented a tool for automating the analysis, and we have applied the tool to a number of common protocols [18]: the analyses terminated within a few seconds and provided safety proofs for the correct versions of the protocols while failing to validate the flawed versions. Remarkably, attacks are often easily derivable by an inspection of the causal graph. The only human effort required is to capture the protocol in the dialect of the spi-calculus which is often straightforwardly derivable from the protocol description.

Outline of the paper. Section 2 introduces a dialect of the spi-calculus used for modeling security protocols. Section 3 introduces causal graphs. Section 4 defines the abstract interpretation framework and states the soundness results. Section 5 presents the safety results. Section 6 discusses further related work and Section 7 concludes.

2 ρ-spi Calculus

The ρ-spi calculus [11, 12] derives from the spi calculus [4] and inherits many of the features of *Lysa* [8], a dialect of the spi calculus specifically tailored to the analysis of authentication protocols. The ρ-spi calculus differs from both calculi in several respects: it associates principal identities to processes; it syntactically binds keys to their owners; and it provides new authentication-specific constructs. In this paper, we consider a novel dialect of ρ-spi in which encryptions and decryptions are performed on-the-fly when sending and receiving messages, respectively. This dialect in particular links protocol specifications more tightly to their

Table 1 (Our Dialect of) the ρ-spi Calculus

Name			**Process**		
a	$::=$	I,J,A,B,E	P,Q	$::=$	$\mathsf{new}(n).P$
	$\mid$	n,m,k_{IJ}		$\mid$	$\mathsf{new}^{\mp}(k_I).P$
	$\mid$	k_I^+		$\mid$	$\mathsf{in}(M).P$
	$\mid$	k_I^-		$\mid$	$\mathsf{out}(M).P$
Term				$\mid$	$\mathsf{begin}_N^i(A,I,M).P$
M,N	$::=$	a		$\mid$	$\mathsf{end}_N^i(A,I,M).P$
	$\mid$	x,y,z		$\mid$	$A \triangleright P$
	$\mid$	(M,N)		$\mid$	$P \mid Q$
	$\mid$	$\{M\}_u$		$\mid$	$!P$
				$\mid$	$\mathbf{0}$

Notation: u ranges over names and variables.

informal "graphical" descriptions, which only depict sent and received messages without giving a precise semantics on how messages are parsed and constructed.

2.1 Syntax

The formal syntax of our dialect of the ρ-spi calculus is depicted in Table 1. We presuppose a countable set of *names* partitioned into the set of identities $\mathcal{ID}$, the set of *messages* $\mathcal{M}$, including the set of shared keys $\mathcal{K}$, and the set of public and private keys $\mathcal{K}^{\mp}$. The set $\mathcal{ID}$, ranged over by I and J, is further partitioned into the two sets of *trusted principals* $\mathcal{ID}_{\mathcal{P}}$, ranged over by A and B, and *enemies* $\mathcal{ID}_{\mathcal{E}}$, ranged over by E. The set $\mathcal{K}$ is composed of keys k_{IJ} shared between I and J. The set $\mathcal{K}^{\mp}$ is partitioned into public and private keys, noted $\mathcal{K}^+$ and $\mathcal{K}^-$, respectively. I's key-pair is composed of a public key k_I^+ and a private key k_I^-, related symbolically by k_I. The set $\mathcal{K}_{\mathcal{E}}$ contains the keys of malicious parties, i.e., the ones where some $E \in \mathcal{ID}_{\mathcal{E}}$ occurs as subscript. For convenience, we syntactically bind keys to their owners, thus assuming that keys are already distributed among protocol participants. Notice that this simplification is not restrictive since, if needed, processes can exchange keys thus modelling session-key distribution. Finally, terms can be paired or encrypted with other terms. [1]

Processes (or *protocols*), ranged over by P and Q, behave as follows: $\mathsf{new}(n).P$ generates a fresh name n local to P while $\mathsf{new}^{\mp}(k_I)$ generates a fresh key-pair for I composed of k_I^+ and k_I^-. We presuppose a unique unnamed public channel, the network, from/to which all principals, including intruders, read and send messages. Similarly to *Lysa*, our input primitive may atomically test part of the read message, by employing pattern-matching. If the input term matches the input pattern, then the variables occurring in the pattern are bound to the remaining sub-part of the

[1]For the sake of readability, in the rest of the paper we omit brackets: for instance, the nested pair $((a,b),k)$ is written as a,b,k.

term; otherwise the term is not read at all. This mechanism is also used to decrypt received messages on-the-fly and thus constitutes an important novelty compared to the ρ-spi calculus; of course, in order to immediately match a term encrypted with asymmetric cryptography, the correct decryption key has to be specified in the pattern. For giving the intuition of the semantics, which is formally defined in Section 2.2, the process '$\mathsf{in}(n).P$' tries to read a specific name n from the network and, if such a name can be read, no binding occurs and P is executed. This is useful, e.g., to check protocols where nonce n is sent encrypted as challenge and received back in clear as response. As another example, consider '$\mathsf{in}(\{x\}_{k_A^-}).P$'. This process reads any ciphertext of the form $\{a\}_{k_A^+}$, decrypts it on the fly, and binds all the free occurrences of x to a in process P. We remark that the key specified in the input pattern is the *decryption* key since for binding x to a the process has to perform a decryption and thus to know the correct decryption key. For easing the presentation, we only consider the instantiation of variables with names: as in [14, 27], we assume that messages are typed so as to distinguish names from the other terms like pairs and ciphertexts.

The $\mathsf{begin}_N^i(A,B,M)$ and $\mathsf{end}_N^i(B,A,M)$ primitives express the *correspondence assertions* [33] in a nonce handshake between A and B. The index i allows for specifying which begin assertion should match a specific end assertion: we assume that the same index is used for at most one begin assertion and one end assertion. The former primitive declares that A is starting a protocol session with B, while the latter declares that B is ending a protocol session in which he believes to have correctly authenticated A: N is the nonce used in the protocol session and M is the authenticated message. Finally, $A \triangleright P$ represents principal A executing process P; $P|Q$ is the parallel composition of P and Q; $!P$ indicates an arbitrary number of parallel instances of P, and $\mathbf{0}$ is the null process that does nothing. In the rest of the paper, we will often omit $\mathbf{0}$ from protocol specifications.

Example 1 Let us consider a simple protocol where B encrypts message m, nonce n and his own identifier with A's public-key and A acknowledges the reception of the first message by sending back the nonce in clear on the network. The specification in ρ-spi calculus is reported below: for the sake of readability, we depict message exchanges in between the corresponding inputs and outputs.

$$
\begin{array}{lcl}
\mathit{Resp} \triangleq & & \mathit{Init} \triangleq \\
 & & \mathsf{new}(m).\mathsf{new}(n). \\
\mathsf{in}(\{B,x,z\}_{k_A^-}). & \xleftarrow{\{B,n,m\}_{k_A^+}} & \mathsf{out}(\{B,n,m\}_{k_A^+}). \\
\mathsf{begin}_x^1(A,B,z). & & \\
\mathsf{out}(x) & \xrightarrow{n} & \mathsf{in}(n). \\
 & & \mathsf{end}_n^1(B,A,m)
\end{array}
$$

$$\mathit{Prot} \triangleq \mathsf{new}^{\mp}(k_A).(B \triangleright !\mathit{Init} \mid A \triangleright !\mathit{Resp})$$

The begin assertion says that A confirms the reception from B of message m in a protocol session based on nonce n and, similarly, the end assertion says that B authenticates A receiving m in a protocol session based on n. The goal of this protocol is to guarantee the secrecy of m and the authentication of A with B.

2.2 Operational semantics

Following [9], the ρ-spi calculus comes with a trace-based semantics. Each process primitive has an associated action and we denote with Act the set of all possible actions. The dynamics of the calculus is formalized by means of a transition relation between *configurations*, i.e., pairs $\langle s,P\rangle$, where $s \in \mathit{Act}^*$ is a trace and P is a closed process. In the following, ε denotes the empty trace. Each transition $\langle s,P\rangle \to \langle s::t,P'\rangle$ simulates one computation step in P and records the corresponding action t in the trace. We denote by $\to^*$ a finite sequence of computation steps. In the following, G ranges over ground terms, namely terms containing no variable. Principals do not directly synchronize with each other. Instead, they may receive from the unique channel an arbitrary message known to the environment, which models the Dolev-Yao intruder [17]: the knowledge of the environment associated with a ρ-spi trace, characterized by the judgement $s \vdash G$, is formalized by a set of deduction rules stating that the environment knows all the messages sent on the network, every message which is not restricted in the trace, all public keys, and enemies' keys. The environment can also construct/destruct pairs and encrypt/decrypt ciphertexts if the appropriate key is known. Formal definitions are postponed to Appendix A.

Definition 1 (Traces) *The set $T(P) \triangleq \{s \mid \exists P'$ s.t. $\langle \varepsilon,P\rangle \to^* \langle s,P'\rangle\}$ is the set of all the traces generated by a finite sequence of transitions from the configuration $\langle \varepsilon,P\rangle$.*

Secrecy and authenticity are defined in terms of ρ-spi traces and processes.

Definition 2 (Secrecy) *A trace s guarantees secrecy of G if and only if $s \nvdash G$. A process P guarantees secrecy of G if and only if $s \nvdash G$ for all $s \in T(P)$.*

The weak authenticity property refines the standard *non-injective agreement* property of [28, 33] by making explicit the nonce used in the handshake and the index binding the begin assertion with the corresponding end assertion.

Definition 3 (Weak Authenticity) *A trace s guarantees weak authenticity if and only if $s = s_1 :: end_{G_1}^i(B,A,G_2) :: s_2$ implies $begin_{G_1}^i(A,B,G_2) \in s_1$. A process P guarantees weak authenticity if and only if s guarantees weak authenticity for all $s \in T(P)$.*

Intuitively, this guarantees that whenever B authenticates A and the message G_2 in an handshake with the nonce G_1, then A engaged in a protocol session with nonce G_1 for authenticating G_2 with B. Similarly to the *agreement* property of [28, 33], the notion of strong authenticity requires the freshness of authentication requests.

Definition 4 (Strong Authenticity) *A trace s guarantees strong authenticity if and only if whenever $s = s_1 :: end^i_{G_1}(B,A,G_2) :: s_2$, we have that $s_1 = s'_1 :: begin^i_{G_1}(A,B,G_2) :: s''_1$ and $s'_1 :: s''_1 :: s_2$ guarantees strong authenticity. A process P guarantees strong authenticity if and only s guarantees strong authenticity for all $s \in T(P)$.*

2.3 Notational conventions

For easing the presentation of the static analysis technique, we use a number of notational conventions. The message restriction $\mathsf{new}(n).P$ is a binder for the message n, the key-pair restriction $\mathsf{new}^{\mp}(k_I).P$ is a binder for k_I^+ and k_I^-, namely the key-pair based on k_I, and the input primitive is a binder for the variables occurring in the input term with the exception of the decryption keys. In all cases the scope of the binders is the continuation process. Similarly, $new(n)$ is a binder for message n and $new^{\mp}(k_I)$ is a binder for the key-pair based on k_I and their scope is the continuation trace. The names and variables occurring free and bound in processes and traces are defined as usual. As in companion transition systems, e.g. [10], we implicitly identify processes up to renaming of bound variables and names, i.e., up to α-equivalence. To simplify the definition of the static analysis and following [8], we discipline the α-renaming of bound names. We stipulate that for each message n there is a *canonical representative*, noted $\lfloor\!\lfloor n \rfloor\!\rfloor$, and we demand that two messages are α-convertible only when they have the same canonical representative. Similar assumption and notation applies to key-pairs and variables. We write $\lfloor\!\lfloor P \rfloor\!\rfloor$ to denote the process obtained by replacing each name and variable in P by the corresponding canonical representative. For convenience, we also require that keys, both symmetric and asymmetric ones, having the same canonical representative depend on the same identifiers, i.e., they have the same subscripts. We assume that the set $\mathcal{M}_{\mathcal{E}}$ of names generated by the environment (cf. rule ENV in Table 7) has a canonical representative which is different from the ones of the names bound in the process. Finally, we assume that the bound names of a process are renamed apart and that they do not clash with the free names; much in the same way variables are assumed to be all distinct. For convenience and without loss of generality, we shall reason on protocol specifications in which every bound name and variable has a different canonical representative.

3 Causal Graphs

A *causal graph* $\mathscr{C} = (\mathcal{N}, \mathcal{E})$ is a finite directed graph with nodes $\mathcal{N}$ and edges $\mathcal{E}$. In the following, we will often write *nodes*$(\mathscr{C})$ to denote the set of nodes in $\mathscr{C}$ and *edges*$(\mathscr{C})$ to denote the set of edges in $\mathscr{C}$. Nodes represent abstractions of process events, while edges track the causality among process events. In the following, we discuss and formalize each of these components.

3.1 Nodes and edges

Nodes, also referred to as abstract processes and ranged over by P and Q, are ρ-spi processes built upon ρ-spi names and the following new name categories:

- *message* $\mathscr{E}$ abstracting messages generated by the environment, identities, public-keys and attackers' keys;
- *labelled names* $\mathsf{a}_{(\mathsf{x})}$, which are semantically equivalent to names but allow for tracking variable instantiation. As an example, the instantiation in P of the variable x by the name n yields $\mathsf{P}[\mathsf{n}_{(\mathsf{x})}/\mathsf{x}]$. In fact, labelling is local to sequential processes and does not propagate through concurrent threads.
- *indexed names* $\mathsf{a}^{(\mathsf{out}(\mathsf{G}).\mathsf{P},i)}$, where $\mathsf{out}(\mathsf{G}).\mathsf{P}$ is the node outputting a and i the positional index of a in G: they only occur within the environment's knowledge and give a precise characterization of which names have been used by the environment to construct a certain term and, notably, the place (i.e., node and position) in which they are sent on the network.

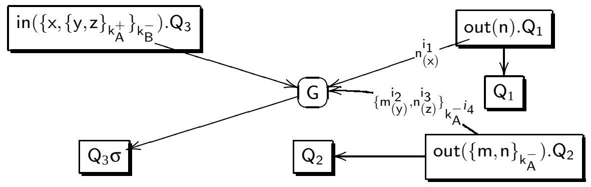
For distinguishing ρ-spi terms from the ones used in causal graphs, we write the latter by $\mathsf{sanserif}$ fonts. In the following, we refer to the set of possibly labelled and indexed names as abstract names. Furthermore, we let G range over ground terms, M,N over terms possibly containing variables, v over abstract names, u over abstract names and variables and $\mathcal{G}$ over ground term sets.

Edges connect two nodes, so representing the causality among such nodes. We discern two kinds of causality, namely *intra-thread* and *inter-thread causality*.

Intra-thread causality links nodes within the same thread and is represented by edges of the form $\mathsf{P} \rightarrow \mathsf{Q}$, linking P to the process Q obtained by reducing P.

Inter-thread causality is due to the synchronization among the process P inputting G and reducing into Q and the nodes $\mathsf{out}(\mathsf{G}_i).\mathsf{P}_i$ outputting the terms used by the environment to construct G. We call such terms integer components since they are not forged by the environment but simply forwarded. They are crucial for the soundness of our abstraction, which identifies unbounded protocol sessions into a finite model: the soundness of the security properties proved on top of this abstraction requires to determine when different protocol sessions may be interleaved. This may happen when the environment exploits terms output in different protocol sessions to construct a term that is sent to a principal engaging in another protocol session: in fact, messages within the same integer component prove to abstract messages belonging to the same protocol session, while messages in different integer components may abstract messages belonging to different protocol sessions. This kind of causality is represented by (*i*) *input edges* of the form $\mathsf{in}(\mathsf{M}).\mathsf{P}\!\!-\!\!\boxed{\mathsf{G}}\!\rightarrow \mathsf{Q}$, connecting the input process with the process obtained by performing the input via the synchronization point $\boxed{\mathsf{G}}$; and (*ii*) *output edges* of the form $\mathsf{out}(\mathsf{G}_i) \xrightarrow{\mathsf{G}'_i} \boxed{\mathsf{G}} \rightarrow \mathsf{Q}$, one for each integer component G'_i of the input term. These edges are labelled by the integer component and connect the process outputting such a component to the process obtained by performing the input via the synchronization point $\boxed{\mathsf{G}}$. A synchronization point links an input edge to the output edges required to construct the input term.

Example 2 Let $\mathscr{C}$ be a causal graph containing the nodes $\mathsf{out}(\mathsf{n}).\mathsf{Q}_1$, $\mathsf{out}(\{\mathsf{m},\mathsf{n}\}_{\mathsf{k}_\mathsf{A}^-}).\mathsf{Q}_2$ and $\mathsf{in}(\{\mathsf{x},\{\mathsf{y},\mathsf{z}\}_{\mathsf{k}_\mathsf{A}^+}\}_{\mathsf{k}_\mathsf{B}^-}).\mathsf{Q}_3$. Let us define the following indexes: $i_1 = (\mathsf{out}(\mathsf{n}).\mathsf{Q}_1, 1)$, $i_2 = \mathsf{out}(\{\mathsf{m},\mathsf{n}\}_{\mathsf{k}_\mathsf{A}^-}).\mathsf{Q}_2, 1)$, $i_3 = \mathsf{out}(\{\mathsf{m},\mathsf{n}\}_{\mathsf{k}_\mathsf{A}^-}).\mathsf{Q}_2, 2)$, and $i_4 = \mathsf{out}(\{\mathsf{m},\mathsf{n}\}_{\mathsf{k}_\mathsf{A}^-}).\mathsf{Q}_2, 3)$. The environment may combine the two output terms into $\{\mathsf{n}^{i_1}, \{\mathsf{m}^{i_2}, \mathsf{n}^{i_3}\}_{\mathsf{k}_\mathsf{A}^-{}^{i_4}}\}_{\mathscr{E}}$, which matches the input pattern $\{\mathsf{x},\{\mathsf{y},\mathsf{z}\}_{\mathsf{k}_\mathsf{A}^+}\}_{\mathsf{k}_\mathsf{B}^-}$ with substitution $\sigma = [\mathsf{n}_{(\mathsf{x})}/\mathsf{x}, \mathsf{m}_{(\mathsf{y})}/\mathsf{y}, \ \mathsf{n}_{(\mathsf{z})}/\mathsf{z}]$: the substitution σ is used to instantiate the free variables in the process Q_3 following the input pattern. Thus $\mathscr{C}$ contains the following nodes and edges, where $\mathsf{G} = \{\mathsf{n}_{(\mathsf{x})}{}^{i_1}, \{\mathsf{m}_{(\mathsf{y})}{}^{i_2}, \mathsf{n}_{(\mathsf{z})}{}^{i_3}\}_{\mathsf{k}_\mathsf{A}^-{}^{i_4}}\}_{\mathscr{E}}$:



Notice that the synchronization point tracks both the positional indexes of the names occurring in the input term and the variable assignment induced by the input pattern.

3.2 Causal graphs as abstraction of ρ-spi processes

Before formalizing the causal graph generation, we introduce some useful notations. We write $\lfloor \mathsf{M} \rfloor$, $\lceil \mathsf{M} \rceil$ and $[\mathsf{M}]$ to denote the term obtained from M by label erasure, index erasure, and both label and index erasure, respectively.

Function $\mathit{outmsg}(\mathsf{out}(\mathsf{G}).\mathsf{P})$ yields the term obtained from G by erasing the labels in G and indexing each name a in G with the pair composed of process $\mathsf{out}(\mathsf{G}).\mathsf{P}$ and the positional index of a in G. This function is naturally extended to node sets, thus characterizing the set of output terms: more formally, $\mathit{outmsg}(\mathcal{N}) = \{\mathit{outmsg}(\mathsf{out}(\mathsf{G}).\mathsf{P}) \mid \mathsf{out}(\mathsf{G}).\mathsf{P} \in \mathcal{N}\}$. Notice that labels are local and do not propagate into the environment's knowledge.

The knowledge of the abstract environment is formalized by the judgement $\mathcal{G} \vdash \mathsf{G}$ (cf. Table 8 of Appendix B), meaning that the environment can construct G given the knowledge of the terms in $\mathcal{G}$. The environment knows every term in $\mathcal{G}$ and the special name $\mathscr{E}$, it can construct and destruct pairs and encrypt and decrypt terms provided that it knows the encryption and decryption keys, respectively. For reducing the number of terms known to the environment, we prevent the environment from deriving identities, public keys and enemies' keys, which abstract the same ρ-spi terms as $\mathscr{E}$. Function $\mathsf{bind} : (\mathsf{M},\mathsf{G}) \mapsto (\mathsf{G}',\sigma)$, reported in Table 8 of Appendix B, defines the pattern-matching among terms. This function takes as input a term M and a ground term G and, if the two terms match, yields the term G' obtained by labelling G according to the variables in M and the substitution σ expressing the variable instantiation induced by the pattern-matching. If pattern-matching fails, bind returns $\uparrow$. For example, $\mathsf{bind}(\mathsf{x},\mathsf{n}^5) = (\mathsf{n}^5_{(\mathsf{x})}, [\mathsf{n}_{(\mathsf{x})}/\mathsf{x}])$. Function *thread*, defined below, yields the set of threads in P, abstracting away from identifiers and replications.

$$\mathit{thread}(\mathsf{P}) \triangleq \begin{cases} \mathit{thread}(\mathsf{P}_1) \cup \mathit{thread}(\mathsf{P}_2) & \text{if } \mathsf{P} = \mathsf{P}_1 | \mathsf{P}_2 \\ \mathit{thread}(\mathsf{Q}) & \text{if } \mathsf{P} \in \{\mathsf{A} \triangleright \mathsf{Q}, !\mathsf{Q}\} \\ \{\mathsf{P}\} & \text{otherwise} \end{cases}$$

Function $\mathit{int}(\mathcal{N}, \mathsf{G})$ yields the set of integer terms in G, that is the largest subterms of G that have not been generated by the environment, each of them paired with the outputting process. In fact, integer terms are either names or ciphertexts whose encryption key is unknown to the environment.

$$\mathit{int}(\mathcal{N}, \mathsf{G}) \triangleq \begin{cases} \{(\mathsf{v}^{(\mathsf{P},\mathsf{j})}, \mathsf{P})\} & \text{if } \mathsf{G} = \mathsf{v}^{(\mathsf{P},\mathsf{j})} \\ \emptyset & \text{if } \mathsf{G} = \mathscr{E} \\ \mathit{int}(\mathcal{N}, \mathsf{G}_1) \cup \mathit{int}(\mathcal{N}, \mathsf{G}_2) & \text{if } \mathsf{G} = (\mathsf{G}_1, \mathsf{G}_2) \\ \{(\mathsf{G}, \mathsf{P})\} & \text{if } \mathsf{G} = \{\mathsf{G}'\}_{\mathsf{v}^{(\mathsf{P},\mathsf{j})}} \wedge \\ & \quad \mathit{outmsg}(\mathcal{N}) \nvdash \mathsf{v}^{(\mathsf{P},\mathsf{j})} \\ \mathit{int}(\mathcal{N}, \mathsf{G}') \cup \mathit{int}(\mathcal{N}, \mathsf{v}') & \text{if } \mathsf{G} = \{\mathsf{G}'\}_{\mathsf{v}'} \wedge \\ & \quad \mathit{outmsg}(\mathcal{N}) \vdash \mathsf{v}' \end{cases}$$

The next definition characterizes the causal graph associated with an abstract process. Here and throughout this paper, we write $\mathsf{P} =_\alpha \mathsf{Q}$ to say that P and Q are α-equivalent and $\llbracket \mathsf{P} \rrbracket$ to denote the process obtained by replacing each name and variable in P with the corresponding canonical representative.

Definition 5 (Causal Graph) *The causal graph $(\mathcal{N}, \mathcal{E})$ associated with P, written as $(\mathcal{N}, \mathcal{E}) = graph(P)$, is given by the least $\mathcal{N}$, $\mathcal{E}$ satisfying the following conditions:*

(*i*) $thread(P) \subseteq \mathcal{N}$

(*ii*) $\mathsf{p.P} \in \mathcal{N} \wedge \mathsf{Q} \in thread(\mathsf{P}) \wedge \mathsf{p} \neq \mathsf{in}(\cdot)$
$\Rightarrow \mathsf{Q} \in \mathcal{N} \wedge \mathsf{p.P} \rightarrow \mathsf{Q} \in \mathcal{E}$

(*iii*) $\mathsf{in(M).P} \in \mathcal{N} \wedge outmsg(\mathcal{N}) \vdash \mathsf{G} \wedge \mathsf{bind(M,G)} = (\mathsf{G}', \sigma)$
$\wedge \mathsf{P}' \in thread(\mathsf{P}\sigma) \wedge (\mathsf{G}'_1, \mathsf{out(G_1).P_1}) \in int(\mathcal{N}, \mathsf{G}')$
$\Rightarrow \exists \mathsf{Q} \in \mathcal{N}$ *s.t.* $\mathsf{Q} =_\alpha \mathsf{P}' \wedge \mathsf{in(M).P} \text{—} \boxed{\mathsf{G}'} \rightarrow \mathsf{Q} \in \mathcal{E}$
$\wedge \mathsf{out(G_1).P_1} \overset{\mathsf{G}'_1}{\text{—}} \boxed{\mathsf{G}'} \rightarrow \mathsf{Q} \in \mathcal{E}$

(*iv*) $\forall \{\mathsf{new(n).P}, \mathsf{new(m).Q}\} \subseteq \mathcal{N}$,
$\lfloor\!\lfloor \mathsf{new(n).P} \rfloor\!\rfloor = \lfloor\!\lfloor \mathsf{new(m).Q} \rfloor\!\rfloor \Leftrightarrow \mathsf{n} = \mathsf{m}$

Function $graph : P \mapsto (\mathcal{N}, \mathcal{E})$ is in fact a closure operator on causal graphs ordered by inclusion. In particular, (*i*) the threads in P are part of the set $\mathcal{N}$ of nodes; (*ii*) if $\mathsf{p.P}$ is a node in the causal graph, with $\mathsf{p} \neq \mathsf{in}(\cdot)$, then the processes in $thread(\mathsf{P})$, which are connected to $\mathsf{p.P}$ by direct edges, belong to the set of nodes; and (*iii*) if $\mathsf{in(M).P}$ is a node and the environment knows a term G matching M with substitution σ, then the set of nodes contains the processes in $thread(\mathsf{P}\sigma)$, up to α-equivalence, and the set of edges contains the input edges connecting $\mathsf{in(M).P}$ to the processes in $thread(\mathsf{P}\sigma)$ and the output edges linking the output processes used by the environment to construct G to the processes in $thread(\mathsf{P}\sigma)$. Intuitively, the input of a term is preceded by the output of *all* messages required by the environment to construct such a term. This point will be clarified later on, when formalizing the causality relation expressed by causal graphs. Condition (*iv*) rules the α-renaming of bound names possibly introduced in the causal graph by condition (*iii*). This α-renaming allows the abstract domain to distinguish the names generated in different threads, thus making the analysis more precise. Unfortunately, this may introduce an infinite number of names in the knowledge of the environment and possibly an infinite number of nodes in the causal graph. We tackle this problem by requiring that whenever two restriction nodes occur in the causal graph and the restricted names differ because of α-renaming, then at least one variable is instantiated with names belonging to different equivalence classes in the two continuation processes. The idea is to apply α-renaming of bound names in the processes following an input for every different input term, where the difference among input terms is deemed up to name equivalence classes. This enhances the precision of the analysis, yet guaranteeing the finiteness of the causal graph.

We say that two graphs are isomorphic when they are obtained by α-renaming of bound names. As stated by the following proposition, every abstract process admits a unique causal graph up to isomorphism. In the following, we write $length(P)$ to denote the number of primitives in P and $length(\mathcal{N})$ to denote the maximal length of the processes in $\mathcal{N}$, namely the number n such that $length(\mathsf{Q}) = n$, for some $\mathsf{Q} \in \mathcal{N}$, and, for every $\mathsf{Q}' \in \mathcal{N}$, $length(\mathsf{Q}') \leq n$.

Proposition 1 (Uniqueness) *For every process P, there exists a unique graph(P) up to isomorphism.*

Proof. Consider the function $\Phi_P(\mathcal{N}, \mathcal{E})$ yielding the least $\mathcal{N}', \mathcal{E}'$ satisfying the three conditions of Definition 5. For proving the thesis, we prove that, for every $\mathcal{N}$ and $\mathcal{E}$, there exists a unique least fixpoint of $\Phi_P(\mathcal{N}, \mathcal{E})$: this trivially implies the thesis. Notice that Φ_P is monotonous over node and edge sets ordered by inclusion.

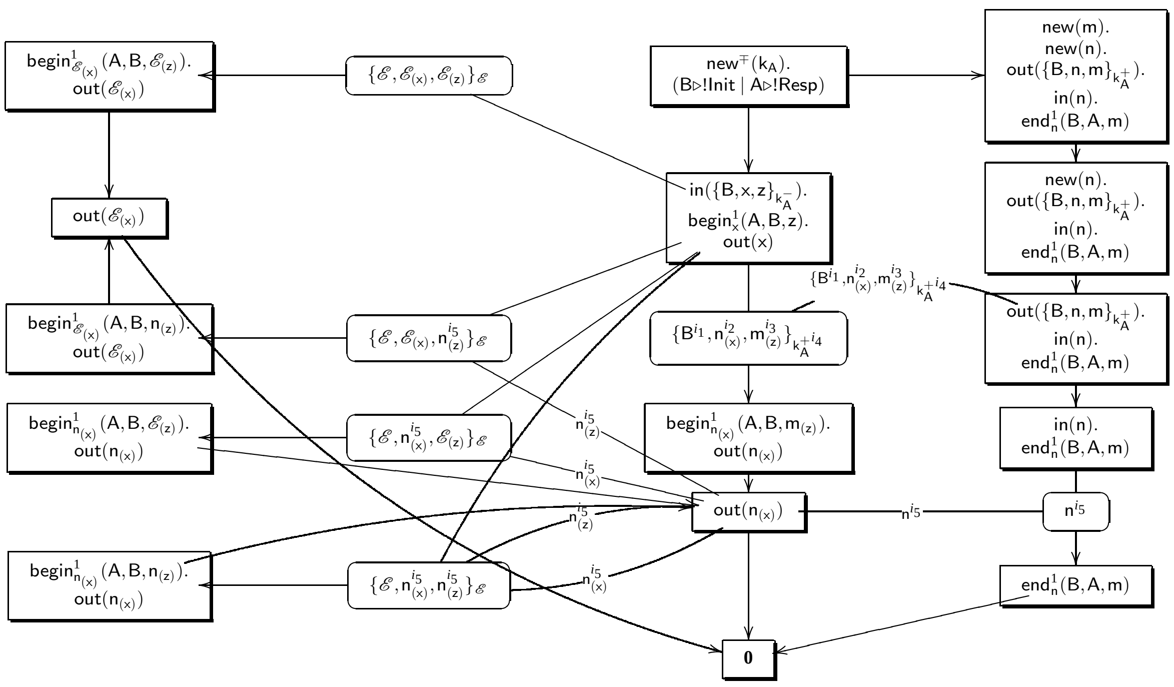
It is easy to see that if $\Phi_P(\mathcal{N}, \mathcal{E}) = \mathcal{N}', \mathcal{E}'$, then $length(\mathcal{N}) = length(\mathcal{N}')$. The set of variables occurring in the processes of $\mathcal{N}'$ is contained into the set of variables occurring in the processes of $\mathcal{N}$. The situation for names is different, since some variable might be instantiated with $\mathscr{E}$ and, notably, new names might be introduced by α-renaming (condition (*iii*)). However, because of condition (*iv*), the number of α-renamed copies of each name is bound by N^X, where N and X are the number of restrictions and variables in $\mathcal{N}$, respectively. The set of processes of finite length and composed of a finite set of names and variables is finite, as well as the set of edges. By Knaster-Tarski theorem, Φ_P has a unique least fixpoint. □

The following corollary says that the size of the graph grows exponentially with the protocol *specification*, which is however fixed in advance, regardless of the number of considered sessions and the protocol run-time behavior. We remark that this result refers to the worst case: in practice, the number of nodes for the protocols we have considered so far does not exceed 100 nodes and the analysis always terminated within a few seconds.

Corollary 1 (Size of causal graphs) *Let $\mathscr{C}$ be a causal net and P a process such that $\mathscr{C} = graph(P)$. Let N and X be the number of restrictions and variables in P, respectively. Then $|nodes(\mathscr{C})| = O(length(P) * N^{X^2})$.*

Example 3 The causal graph associated with protocol *Prot* of Example 1 is depicted in Table 2. The rounded boxes represent synchronization points while the other ones abstract processes. The analysis of the causal graph gives us some interesting information about the run-time behavior of protocol participants. Even if the protocol is simple, it turns out to be interesting since the attackers know both the public-key used for encrypting the first message and, after the second message, the nonce used in the protocol session. This increases the number of actions at their disposal and, consequently, the number of nodes and edges in the causal graph. In general, fixed the number of message exchanges, the causal graph for protocols preserving the secrecy of

Table 2 Causal graph *graph*(*Prot*)



new(m). new(n). out({B,n,m}_{k_A^+}). in(n). end_n^1(B,A,m)
new^∓(k_A). (B▷!Init | A▷!Resp)
in({B,x,z}_{k_A^-}). begin_x^1(A,B,z). out(x)
new(n). out({B,n,m}_{k_A^+}). in(n). end_n^1(B,A,m)
out({B,n,m}_{k_A^+}). in(n). end_n^1(B,A,m)
in(n). end_n^1(B,A,m)
end_n^1(B,A,m)
out(n_(x))
n^{i5}
0
out(𝒮_(x))

Legend: for every $j \in [1,4]$, i_j= $(\mathsf{out}(\{\mathsf{B},\mathsf{n},\mathsf{m}\}_{\mathsf{k}_\mathsf{A}^+}).\mathsf{in}(\mathsf{n}).\mathsf{end}^1_\mathsf{n}(\mathsf{B},\mathsf{A},\mathsf{m}),j)$; i_5=$(\mathsf{out}(\mathsf{n}_{(\mathsf{x})}),1)$

messages and relying on digital signature or symmetric-key cryptography is typically simpler than the one of protocols based on public-key cryptography and possibly exposing some messages to the attackers. Let $\mathcal{N}$ be the set of nodes in the causal graph: we can see that $outmsg(\mathcal{N}) \nvdash \mathsf{m}^i$ for any index i and this intuitively means that the authenticated message is kept secret, as expected. Furthermore, the only process asserting an end event is $\mathsf{end}^1_\mathsf{n}(\mathsf{B},\mathsf{A},\mathsf{m})$. This is preceded by the output of $\mathsf{n}_{(\mathsf{x})}$ (inter-thread causality), which is in turn preceded (intra-thread causality) by node $\mathsf{begin}^1_{\mathsf{n}_{(\mathsf{x})}}(\mathsf{A},\mathsf{B},\mathsf{m}_{(\mathsf{z})}).\mathsf{out}(\mathsf{n}_{(\mathsf{x})})$. Note that the output of $\mathsf{n}_{(\mathsf{x})}$ may be also preceded (intra-thread causality) by $\mathsf{begin}^1_{\mathsf{n}_{(\mathsf{x})}}(\mathsf{A},\mathsf{B},\mathscr{E}_{(\mathsf{z})}).\mathsf{out}(\mathsf{n}_{(\mathsf{x})})$ and $\mathsf{begin}^1_{\mathsf{n}_{(\mathsf{x})}}(\mathsf{A},\mathsf{B},\mathsf{n}_{(\mathsf{z})}).\mathsf{out}(\mathsf{n}_{(\mathsf{x})})$. These events are enabled by the environment forging the messages $\{\mathscr{E},\mathsf{n}^{i_5},\mathscr{E}\}_\mathscr{E}$ and $\{\mathscr{E},\mathsf{n}^{i_5},\mathsf{n}^{i_5}\}_\mathscr{E}$, respectively (cf. the corresponding synchronization points), and sending them to the responder. This may happen only if the environment knows n and these processes are thus preceded (inter-thread causality) by the output of $\mathsf{n}_{(\mathsf{x})}$. This cycle in the graph tells us that the environment can actively interact with the responder only after the responder has received the message generated by the initiator and "asserted" $\mathsf{begin}^1_{\mathsf{n}_{(\mathsf{x})}}(\mathsf{A},\mathsf{B},\mathsf{m}_{(\mathsf{z})})$, according to the intended protocol run. Intuitively, this means that the causal graph guarantees authenticity. Finally, the portion of the graph in the upper-left corner shows that the environment can forge messages in which the message $\mathscr{E}$ replaces the nonce, but the resulting processes do not increase the environment's knowledge as they eventually output $\mathscr{E}$, which is already known to the environment.

3.3 Paths

Edges describe the causality among process events and each edge may be naturally associated with an action: for instance, if a causal graph contains the edge $\mathsf{out}(\mathsf{n}).\mathsf{P} \to \mathsf{P}$, then the process P is causally preceded by the reduction of the process $\mathsf{out}(\mathsf{n}).\mathsf{P}$ with action $out(\mathsf{n})$. In general, we express the causality dependency among process events through *paths*, which are sequences of *abstract actions* expressing process events. Abstract actions extend ρ-spi ones with the new action $in_{com}(\mathsf{G}_1,\mathsf{G}_2)$, which is used for tracking inputs involved in inter-thread causality: G_1 is the integer component and G_2 is the input pattern, in which variables are replaced with the actual received messages.

We now argue on the number of paths associated with a node. We have mentioned that the input of a term is causally preceded by the outputs of *all* the integer components needed by the environment to construct such a term. Thus an input is preceded by a set of paths. However, the same process may be generated by the input of different terms, say G_1 and G_2. Then such a process is preceded by either all the outputs needed by the environment to construct G_1 or all the outputs needed to construct G_2. This is the reason why the paths associated with a node are actually a set of path sets, meaning that the node is causally preceded by *all* the paths in *one* of its path sets. Notice that, as result of our abstraction which essentially collapses an unbounded number of instances of each principal into a strand of nodes related by intra-thread causality, causal graphs may contain cycles. For this reason, when evaluating the paths preceding a node, we need to avoid loops: this is achieved by traversing input edges only once, thus abstracting away from cycles in the causal graph which, in fact, do not alter the causality among nodes. In the following, we write $\{s_1, \dots, s_n\} :: \tau$ to denote $\{s_1 :: \tau, \dots, s_n :: \tau\}$ and $outedg(G, Q, \mathscr{C})$ to denote the set of output edges in $\mathscr{C}$ incoming in Q via the synchronization point $\boxed{G}$. Function $paths : \mathscr{C}, P, \mathcal{E} \mapsto \mathscr{S}$, defined below, yields the set $\mathscr{S}$ of paths sets that are associated with the node P and do not traverse the edges in $\mathcal{E}$. We often write $paths(\mathscr{C}, P)$ to denote $paths(\mathscr{C}, P, \emptyset)$.

Definition 6 (Paths) *Let $\mathscr{C}$ be a causal graph and $Q \in nodes(\mathscr{C})$. The paths preceding Q in $\mathscr{C}$ and not traversing the edges in $\mathcal{E}$, written $paths(\mathscr{C}, Q, \mathcal{E})$, are defined as the least $\mathscr{S}$ such that*

(a) Q *has no incoming edge* $\Rightarrow \mathscr{S} = \{\emptyset\}$

(b) $p.P \rightarrow Q \in edges(\mathscr{C}) \wedge \mathcal{S} \in paths(\mathscr{C}, p.P, \mathcal{E}) \Rightarrow \mathcal{S} :: p \in \mathscr{S}$

$$
\begin{aligned}
(c)\ & \mathsf{in}(M).P\text{—}\boxed{G}\rightarrow Q \in edges(\mathscr{C}) \setminus \mathcal{E} \ \wedge\ \mathsf{bind}(M, \lfloor G \rfloor) = (G, \sigma) \\
& \quad \wedge \bigcup_{i \in [1,n]} \{\mathsf{out}(G_i).P_i \overset{G'_i}{\text{—}}\boxed{G}\rightarrow Q\} = outedg(G, Q, \mathscr{C}) \\
& \quad \wedge\ \mathcal{S} \in paths(\mathscr{C}, \mathsf{in}(M).P, \mathcal{E} \cup \{\mathsf{in}(M).P\text{—}\boxed{G}\rightarrow Q\}) \\
& \quad \wedge\ \mathcal{S}_i \in paths(\mathscr{C}, \mathsf{out}(G_i).P_i, \mathcal{E} \cup \{\mathsf{in}(M).P\text{—}\boxed{G}\rightarrow Q\}) \\
& \Rightarrow\ \mathcal{S}{::}in(M\sigma) \cup \bigcup_{i \in [1,n]} \mathcal{S}_i{::}out(G_i) :: in_{com}(G'_i, M\sigma) \in \mathscr{S}
\end{aligned}
$$

If the node Q is preceded by intra-thread causality by $p.P$ (condition (a)), then the path sets associated with Q are the ones associated with $p.P$, each of them extended with the action p. If Q is obtained by the reduction of process $\mathsf{in}(M).P$ receiving G and the input edge $\mathsf{in}(M).P\text{—}\boxed{G}\rightarrow Q$ is not in $\mathcal{E}$, then the paths associated with Q are the ones associated with $\mathsf{in}(M).P$, each of them extended with the action $in(M\sigma)$, plus the ones associated with the processes outputting the terms used to construct G (edges $\mathsf{out}(G_i).P_i \overset{G'_i}{\text{—}}\boxed{G}\rightarrow Q$), extended with $out(G_i) :: in_{com}(G'_i, M\sigma)$. Notice that the edge $\mathsf{in}(M).P\text{—}\boxed{G}\rightarrow Q$ is inspected only once, thus avoiding loops due to cycles in the graph.

3.4 Names and sessions

Since a causal graph abstracts an unbounded number of protocol sessions, an interesting issue related to our abstraction is that different occurrences in a path of the same name n might actually abstract different ρ-spi names, one for each abstracted protocol session. The problem is that the environment may exploit messages generated in different protocol sessions so as to forge a message which is then used to interact with another session: in fact, this kind of interleaving may break the security goals of the protocol. According to the definition of causal graphs, the labels of the output edges express the integer components of the input term. This information can be used for soundly characterizing which names abstract over the same protocol session and, in particular, which names abstract the same ρ-spi name. Intuitively, it is sufficient to check for every pair of output and input actions related by inter-thread causality which messages belong to the integer component and which do not: the former abstract over the same protocol session while the latter may belong to different protocol sessions. Here and throughout this paper, we write $msgs(M)$ to denote the set of messages in M, including those labelled. We also write $\pi^i(G)$ to denote the name occurring in the i-th position of G and $\pi^i(s)$ to denote the i-th action in s. Finally, $|s|$ denotes the number of actions in s.

Table 3 introduces the deduction system for judgement $s \vdash (v, i) = (v', j)$, meaning that names v and v, occurring in the i-th and j-th action of s respectively, abstract over the same ρ-spi name. Intuitively, the concretization of a path yields the set of ρ-spi traces obtained by instantiating the abstract names occurring therein: if $s \vdash (v, i) = (v', j)$, then the occurrences of v and v' in the i-th and j-th action of s, respectively, are instantiated with the same ρ-spi name (see Section 4 for more detail). Rule Ax says that the occurrences of the same message within a single action abstract the same ρ-spi name. Rule Intra says that the occurrences of a message within actions related by intra-thread causality abstract the same ρ-spi name, since such actions belong to the same thread and thus abstract over the same protocol session. Rule Inter-Bind says that messages received within an integer component, that is messages not manipulated by the attacker, abstract over the same protocol session. Rule Inter-Match says that the pattern-matching of name n allows for recovering a protocol session described in a previous part of the path and, more precisely, it allows to inherit the equality constraints holding at the time of n's restriction. Finally, Trans, Sym, and Prefix make the relation transitive, symmetric, and closed by prefixes, respectively.

Example 4 Let us consider the following path, which represents the intended protocol behavior:

$$
\begin{aligned}
s \ \triangleq\ & new^{\mp}(k_A) :: new(m) :: new(n) :: out(\{B, n, m\}_{k_A^+}) :: \\
& in_{com}(\{B^{i_1}, n_{(x)}{}^{i_2}, m_{(z)}{}^{i_3}\}_{k_A^+}{}^{i_4}, \{B, n_{(x)}, m_{(z)}\}_{k_A^-}) :: \\
& begin^1_{n_{(x)}}(A, B, m_{(z)}) :: out(n_{(x)}) :: \\
& in_{com}(n^{i_5}, n) :: end^i_n(B, A, m)
\end{aligned}
$$

Table 3 Equality among abstract terms

$$\textsc{Ax}\quad \frac{|\mathsf{s} :: \mathsf{t}| = i \qquad \mathsf{v} \in \mathit{msgs}(\mathsf{t})}{\mathsf{s} :: \mathsf{t} \vdash (\mathsf{v}, i) = (\mathsf{v}, i)}$$

$$\textsc{Intra}\quad \frac{|\mathsf{s}| = i \qquad \mathsf{t} \neq \mathit{in}_{com}(\mathsf{G}, \mathsf{G}') \qquad \mathsf{s} \vdash (\mathsf{v}, i) = (\mathsf{v}, j)}{\mathsf{s} :: \mathsf{t} \vdash (\mathsf{v}, i+1) = (\mathsf{v}, i)}$$

$$\textsc{Inter-Bind}\quad \frac{\mathsf{v}^{(\mathsf{P},j)} \in \mathit{msgs}(\mathsf{G}_2) \qquad \pi^j(\mathsf{G}_1) = \mathsf{v}' \qquad |\mathsf{s}| = i}{\mathsf{s}::\mathit{out}(\mathsf{G}_1) :: \mathit{in}_{com}(\mathsf{G}_2, \mathsf{G}) \vdash (\mathsf{v}, i+2) = (\mathsf{v}', i+1)}$$

$$\textsc{Inter-Match}\quad \frac{\begin{array}{c}\mathsf{n} \in \mathit{msgs}(\mathsf{G}) \qquad \mathsf{s} = \mathsf{s}_1::\mathit{new}(\mathsf{n}) :: \mathsf{s}_2 \qquad |\mathsf{s}| = i \\ |\mathsf{s}_1| = j \qquad \mathsf{s} \vdash (\mathsf{n}, i+2) = (\mathsf{n}, j+1) \qquad \mathsf{s}_1 \vdash (\mathsf{v}, j) = (\mathsf{v}', j')\end{array}}{\mathsf{s} :: \mathit{out}(\mathsf{G}_1) :: \mathit{in}_{com}(\mathsf{G}_2, \mathsf{G}) \vdash (\mathsf{v}, i+2) = (\mathsf{v}, j)}$$

$$\textsc{Trans}\quad \frac{\mathsf{s} \vdash (\mathsf{v}, i) = (\mathsf{v}', i') \qquad \mathsf{s} \vdash (\mathsf{v}', i') = (\mathsf{v}'', i'')}{\mathsf{s} \vdash (\mathsf{v}, i) = (\mathsf{v}'', i'')}$$

$$\textsc{Sym}\quad \frac{\mathsf{s} \vdash (\mathsf{v}, i) = (\mathsf{v}', i')}{\mathsf{s} \vdash (\mathsf{v}', i') = (\mathsf{v}, i)} \qquad\qquad \textsc{Prefix}\quad \frac{\mathsf{s} \vdash (\mathsf{v}, i) = (\mathsf{v}', i')}{\mathsf{s} :: \mathsf{t} \vdash (\mathsf{v}, i) = (\mathsf{v}', i')}$$

The messages occurring in the begin and end assertions abstract over the same protocol session. In particular, we can prove $\mathsf{s} \vdash (\mathsf{n}, 9) = (\mathsf{n}_{(\mathsf{x})}, 6)$ via INTRA and INTER-BIND. Similarly, we can prove $\mathsf{s} \vdash (\mathsf{m}, 9) = (\mathsf{m}, 2)$ via INTRA and INTER-MATCH, and $\mathsf{s} \vdash (\mathsf{m}_{(\mathsf{z})}, 6) = (\mathsf{m}, 2)$ via INTRA, INTER-BIND and PREFIX. Finally, by SYMM and TRANS we get $\mathsf{s} \vdash (\mathsf{m}, 9) = (\mathsf{m}_{(\mathsf{z})}, 6)$.

In Section 3.3, we have defined the paths associated to a node in a causal graph and, for avoiding loops due to cycles in the graph, we have requested that input edges are inspected only once. This approximation does not affect the causality among nodes but it might affect the equality constraints. For this reason, we only consider a class of causal graphs, called *cycle-invariant* graphs, for which cycles do not affect the equality constraints of the involved paths.

Definition 7 (Cycle-invariance) *We say that $\mathscr{C}$ is cycle-invariant if and only if for every* $\{\mathsf{in}(\mathsf{M}).\mathsf{P} \!-\!\!\boxed{\mathsf{G}} \rightarrow \mathsf{Q}, \mathsf{out}(\mathsf{G}_1).\mathsf{P}' \xrightarrow{\mathsf{G}_2} \boxed{\mathsf{G}} \rightarrow \mathsf{Q}\} \subseteq \mathit{edges}(\mathscr{C})$, $\mathcal{S}_{out} \in \mathit{paths}(\mathscr{C}, \mathsf{out}(\mathsf{G}_1).\mathsf{P}')$, $\mathcal{S}_{in} \in \mathit{paths}(\mathscr{C}, \mathsf{in}(\mathsf{M}).\mathsf{P})$ *there exists* $\mathcal{S} \in \mathit{paths}(\mathscr{C}, \mathsf{Q})$ *s. t. the following conditions hold:*

- *for every* $\mathsf{s}_1 = \mathsf{s}' :: \mathit{in}(\mathsf{G}') \in \mathcal{S}$ *there exists* $\mathsf{s}_2 \in \mathcal{S}_{in}$ *such that either* $\mathsf{s}_2 = \mathsf{s}'$ *or* $\mathsf{s}_2 = \mathsf{s}' :: \mathit{in}(\mathsf{G}') :: \mathsf{s}'' \in \mathcal{S}_{in}$ *and* $\mathsf{s}_1 \vdash (\mathsf{v}, j) = (\mathsf{v}', j') \Rightarrow \mathsf{s}_2 \vdash (\mathsf{v}, i) = (\mathsf{v}', j')$, *with* $|\mathsf{s}_1| = j$ *and* $|\mathsf{s}_2| = i$.
- *for every* $\mathsf{s}_1 = \mathsf{s}' :: \mathit{out}(\mathsf{G}_1) :: \mathit{in}_{com}(\mathsf{G}_2, \mathsf{G}') \in \mathcal{S}$ *there exists* $\mathsf{s}_2 \in \mathcal{S}_{out}$ *such that either* $\mathsf{s}_2 = \mathsf{s}'$ *or* $\mathsf{s}_2 = \mathsf{s}' :: \mathit{out}(\mathsf{G}_1) :: \mathit{in}_{com}(\mathsf{G}_2, \mathsf{G}') :: \mathsf{s}''$ *and* $\mathsf{s}_1 \vdash (\mathsf{v}, j) = (\mathsf{v}', j') \Rightarrow \mathsf{s}_2 \vdash (\mathsf{v}, i) = (\mathsf{v}', j')$, *with* $|\mathsf{s}_1| = j$ *and* $|\mathsf{s}_2| = i$.

For example, *graph*(*Prot*) is cycle-invariant in that input nodes do not belong to cycles, the only output node within cycles is $\mathsf{out}(\mathsf{n}_{(\mathsf{x})})$, and cycles in the causal graph do not affect the equality constraints on $\mathsf{n}_{(\mathsf{x})}$.

4 Abstract Interpretation

In this section we illustrate the relation between causal graphs and ρ-spi semantics. This is formalized by a concretization function, defined on abstract terms, paths, path sets and causal graphs.

4.1 Concretization of causal graphs

Terms The relation between ρ-spi terms and abstract terms is formally defined in Table 4 by the concretization function $\gamma_{\mathsf{trm}} : \mathsf{M} \mapsto \{M_1, \ldots, M_n\}$. Abstract identities are instantiated with the corresponding ρ-spi identity. The concretization of an abstract message yields the set of messages having the same canonical representative. Similar reasoning applies to variables, public and private keys. The special name $\mathscr{E}$ abstracts over identities, public keys, messages possibly generated by the environment and attackers' keys. Finally, the concretization of the remaining terms is given by instantiating the names and the variables occurring therein.

Paths The concretization function $\gamma_{\mathsf{path}} : \mathsf{s} \mapsto \{(s_1, \sigma_1), \ldots, (s_n, \sigma_n)\}$ takes as input a path and yields a set of pairs composed of a ρ-spi trace s_i and a substitution σ_i from abstract names to ρ-spi names. More precisely, (*i*) each trace is obtained by instantiating the names occurring in each action of the path so as to satisfy the equality constraints associated to the path and (*ii*) the substitution tracks the instantiation of the abstract messages occurring in the actions of the last thread, namely the largest suffix of s related by intra-thread causality. We recall that a path may "cross" different threads and, as a matter of fact, $\mathsf{s} \vdash (\mathsf{v}, k) = (\mathsf{v}', k')$, with k being the length of s, only if v is an abstract name occurring in the last thread.

Example 5 One of the possible concretizations of the path s described in Example 4 is reported below:

$$\begin{array}{l} \mathit{new}^{\mp}(k_A) :: \mathit{new}(m) :: \mathit{new}(n) :: \mathit{out}(\{B, n, m\}_{k_A^+}) :: \\ \mathit{in}(\{B, n, m\}_{k_A^-}) :: \mathit{begin}_n^1(A, B, m) :: \mathit{out}(n) :: \\ \mathit{in}(n) :: \mathit{end}_n^1(B, A, m) \end{array}$$

$\sigma : \mathsf{n} \mapsto n, \ \mathsf{n} \mapsto m$

Table 4 Concretization functions

$$
\gamma_{\mathsf{trm}}(\mathsf{M}) \triangleq \begin{cases}
\{I\} & \text{if } \mathsf{M} = \mathsf{I} \\
\{n \mid \lfloor\!\lfloor n \rfloor\!\rfloor = \lfloor\!\lfloor m \rfloor\!\rfloor\} & \text{if } \mathsf{M} = \mathsf{m} \\
\{y \mid \lfloor\!\lfloor y \rfloor\!\rfloor = \lfloor\!\lfloor x \rfloor\!\rfloor\} & \text{if } \mathsf{M} = \mathsf{x} \\
\{k_I'^{+} \mid \lfloor\!\lfloor k_I' \rfloor\!\rfloor = \lfloor\!\lfloor k_I \rfloor\!\rfloor\} & \text{if } \mathsf{M} = \mathsf{k}_{\mathsf{I}}^{+} \\
\{k_I'^{-} \mid \lfloor\!\lfloor k_I' \rfloor\!\rfloor = \lfloor\!\lfloor k_I \rfloor\!\rfloor\} & \text{if } \mathsf{M} = \mathsf{k}_{\mathsf{I}}^{-} \\
I\mathcal{D} \cup \mathcal{K}^{+} \cup \mathcal{M}_{\mathcal{E}} \cup \mathcal{K}_{\mathcal{E}} & \text{if } \mathsf{M} = \mathscr{E} \\
\gamma_{\mathsf{trm}}(\mathsf{a}) & \text{if } \mathsf{M} \in \{\mathsf{a}_{(\mathsf{x})}, \mathsf{a}^{\mathsf{i}}, \mathsf{a}^{\mathsf{i}}_{(\mathsf{x})}\} \\
\{\mathsf{M}\sigma \mid \forall \mathsf{u} \in dom(\sigma), \sigma(\mathsf{u}) \in \gamma_{\mathsf{trm}}(\mathsf{u})\} & \text{otherwise}
\end{cases}
$$

$$
\begin{aligned}
\gamma_{\mathsf{path}}(\mathsf{s}) &\triangleq \{(s,\sigma) \mid |\mathsf{s}| = |s| = n \wedge \quad (i) \quad \forall i \in [1,n], \exists \sigma_i \text{ s.t. } \pi^i(\bar{\mathsf{s}})\sigma_i = \pi^i(s) \wedge (\mathsf{s} \vdash (\mathsf{v}, j) = (\mathsf{v}', j') \Rightarrow \sigma_j(\mathsf{v}) = \sigma_{j'}(\mathsf{v}')) \wedge \\
&\qquad (ii) \quad (\mathsf{s} \vdash (\mathsf{v}, n) = (\mathsf{v}', k) \wedge \sigma_k(\mathsf{v}') = a) \Leftrightarrow \sigma(\mathsf{v}) = a\} \\
\gamma_{\mathsf{pset}}(\mathcal{S}) &\triangleq \{ (\{s_1, \ldots, s_n\}, \sigma) \mid \mathcal{S} = \{\mathsf{s}_1, \ldots, \mathsf{s}_n\} \wedge \forall i \in [1,n], (s_i, \sigma_i) \in \gamma_{\mathsf{path}}(\mathsf{s}_i) \wedge \sigma = \biguplus_{i=1,n} \sigma_i \neq \uparrow\} \\
\gamma_{\mathsf{net}}(\mathscr{C}) &\triangleq \{\langle s, P\rangle \mid \quad (i) \quad \forall i \in [1,n], \exists \mathsf{Q}_i \in nodes(\mathscr{C}), \mathcal{S}_i \in paths(\mathscr{C}, \mathsf{Q}_i), (S_i, \sigma_i) \in \gamma_{\mathsf{pset}}(\mathcal{S}_i) \text{ s.t. } S_i = S_j :: \pi^i(s), \text{ with } j < i \wedge \\
&\qquad (ii) \quad \forall Q \in thread(P),\ \exists i \in [1,n], \sigma \text{ s.t. } Q = \mathsf{Q}_i\sigma \wedge \sigma \uplus \sigma_i \neq \uparrow \}
\end{aligned}
$$

Notation: $\pi^i(s)$ yields the i-th action in s

$\bar{s}$ yields the trace obtained from s by replacing each occurrence of $in_{com}(\mathsf{G}_1, \mathsf{G}_2)$ with $in(\mathsf{G}_2)$

$\sigma_1 \uplus \sigma_2 = \sigma_1 \cup \sigma_2$ if $\sigma_1 \neq \uparrow \wedge \sigma_2 \neq \uparrow \wedge \mathsf{u} \in dom(\sigma_1) \cap dom(\sigma_2) \Rightarrow \sigma_1(\mathsf{u}) = \sigma_2(\mathsf{u})$

$\sigma_1 \uplus \sigma_2 = \uparrow$ otherwise

Since $\mathsf{s} \vdash (\mathsf{n}_{(\mathsf{x})}, 6) = (\mathsf{n}, 9)$, $\mathsf{n}_{(\mathsf{x})}$ and n are instantiated with the same name n. Similarly, since $\mathsf{s} \vdash (\mathsf{m}_{(\mathsf{z})}, 6) = (\mathsf{m}, 9)$, m and $\mathsf{m}_{(\mathsf{z})}$ are instantiated with the same name m. Notice also that the path guarantees authenticity and the trace guarantees strong authenticity.

Path sets The concretization of a path set is given by the concretization of the paths occurring therein and by the union of the corresponding instantiations. It is worth to mention that the instantiation of the messages in last thread has to be the same ($\biguplus_{i=1,n} \sigma_i \neq \uparrow$): the union $\sigma_1 \uplus \sigma_2$ succeeds only when the abstract names substituted by both σ_1 and σ_2 are bound to the same ρ-spi name.

Configurations The concretization function $\gamma_{\mathsf{net}} : \mathscr{C} \mapsto \{\langle s_1, P_1\rangle,\ \ldots, \langle s_n, P_n\rangle\}$ takes as input a causal graph and yields a set of ρ-spi configurations satisfying two constraints, the former concerning traces and the latter concerning processes. In the following, we let S range over ρ-spi trace sets.

i for every action τ_i in the trace there exists a node Q_i in the causal graph, a path set $\mathcal{S}_i$ associated to Q_i and a trace set S_i such that S_i is an instantiation of $\mathcal{S}_i$ ($(S_i, \sigma_i) \in \gamma_{\mathsf{pset}}(\mathcal{S}_i)$), where S_i is a trace set associated to a preceding action in the trace, extended with τ_i.

ii for every thread Q of the process in the ρ-spi configuration, there exists a node Q_i in the causal graph, a path set $\mathcal{S}_i$, a trace set S_i and an instantiation σ_i such that $Q = \mathsf{Q}_{\mathsf{i}}\sigma$, where σ is compatible with the name instantiation σ_i given by the concretization of the path set associated to τ_i. in the last thread.

Intuitively, condition (i) requires that the actions in the trace respect the causality paths induced by the causal graph and condition (ii) requires that every process has an abstraction in the causal graph and the trace respects the causality paths associated to such a node.

Finally, we state the soundness results of the abstract interpretation. The following theorem says that the causal graph generated from a process is an abstraction of the configuration composed of such a process and the empty trace.

Theorem 1 (Soundness) *If $graph(P) = \mathscr{C}$, then $\langle \varepsilon, P\rangle \in \gamma_{net}(\mathscr{C})$.*

The following theorem says that the set of configurations abstracted by a causal graph is closed under process reduction, i.e., causal graphs are a sound abstraction of the ρ-spi semantics.

Theorem 2 (Preservation) *Let $\mathscr{C}$ be cycle-invariant. If $\langle s, P\rangle \to \langle s', P'\rangle$ and $\langle s, P\rangle \in \gamma_{net}(\mathscr{C})$, then $\langle s', P'\rangle \in \gamma_{net}(\mathscr{C})$.*

5 Safety Results

In this section we state the safety results of our static analysis technique. As stated by the following definition, a causal graph guarantees the secrecy of a name v, if the abstract environment cannot deduce any term which is equal to v up to index erasure.

Definition 8 (Abstract secrecy) *A causal graph $\mathscr{C}$ guarantees the secrecy of* v *if and only if* $outmsg(nodes(\mathscr{C})) \vdash \mathsf{v}'$ *implies* $\lceil \mathsf{v} \rceil \neq \lceil \mathsf{v}' \rceil$.

As formalized below, a path guarantees authenticity if every end assertion is preceded by a corresponding begin assertion and the terms occurring therein are related by equality constraints. This guarantees that authenticity carries over the action sequences abstracted by the path. A causal graph guarantees authenticity if for every $\mathsf{end}^i_{\mathsf{G}_1}(\mathsf{A},\mathsf{B},\mathsf{G}_2).\mathsf{P}$ and $\mathcal{S} \in paths(\mathscr{C}, \mathsf{end}^i_{\mathsf{G}_1}(\mathsf{A},\mathsf{B},\mathsf{G}_2).\mathsf{P})$, there exists a path in $\mathcal{S}$ containing a suitable begin assertion. In the following, we write $\mathsf{G} \simeq_{\mathsf{s},i,j} \mathsf{G}'$, read as G is equivalent to G' in s, if the two terms are equal component-wise, i.e., for every k such that $\pi^k(\mathsf{G}) = \mathsf{v}$ and $\pi^k(\mathsf{G}') = \mathsf{v}'$, we have that v and v' are either the same identity or possibly labelled messages satisfying $\mathsf{s} \vdash (\mathsf{v}, i) = (\mathsf{v}', j)$. This guarantees that the two terms abstract the same ρ-spi term.

Definition 9 (Abstract Authenticity) *A path* s *guarantees authenticity if and only if for every j such that* $\pi^j(\mathsf{s}) = end^i_{\mathsf{G}_1}(\mathsf{A},\mathsf{B},\mathsf{G}_2)$*, there exist* $\mathsf{G}'_1, \mathsf{G}'_2, j' < j$ *s.t.* $\pi^{j'}(\mathsf{s}) = begin^i_{\mathsf{G}'_1}(\mathsf{B},\mathsf{A},\mathsf{G}'_2)$, $\mathsf{G}_1 \simeq_{\mathsf{s},j,j'} \mathsf{G}'_1$ *and* $\mathsf{G}_2 \simeq_{\mathsf{s},j,j'} \mathsf{G}'_2$.

A causal graph $\mathscr{C}$ guarantees authenticity iff for every $\mathsf{end}^i_{\mathsf{G}_1}(\mathsf{A},\mathsf{B},\mathsf{G}_2).\mathsf{P} \in nodes(\mathscr{C})$ *and* $\mathcal{S} \in paths(\mathscr{C}, \mathsf{end}^i_{\mathsf{G}_1}(\mathsf{A},\mathsf{B},\mathsf{G}_2).\mathsf{P})$*, there exists* $\mathsf{s} \in \mathcal{S}$ *s.t.* $\mathsf{s} :: end^i_{\mathsf{G}_1}(\mathsf{A},\mathsf{B},\mathsf{G}_2)$ *guarantees authenticity.*

For example, the path d in Example 4 guarantees authenticity as the end assertion is preceded by a corresponding begin assertion and the messages occurring therein are equivalent. As a matter of fact, every $\mathcal{S} \in paths(graph(\mathsf{Prot}), \mathsf{end}^1_{\mathsf{n}}(\mathsf{B},\mathsf{A},\mathsf{m}))$ contains a path guaranteeing authenticity. Thus *graph*(*Prot*) guarantees authenticity. The following theorems state that causal graphs constitute a sound model for the static verification of secrecy and authenticity. In particular, the next theorem says that if the causal graph associated with P guarantees the secrecy of v, then P guarantees the secrecy of any concretization of v.

Theorem 3 (Secrecy) *Let P be a process and $\mathscr{C} = graph(P)$ a cycle-invariant causal graph. If $\mathscr{C}$ guarantees the secrecy of* v *then P guarantees the secrecy of any* $v \in \gamma_{trm}(\mathsf{v})$.

For instance, the causal graph of Example 3 guarantees the secrecy of m and, consequently, the protocol of Example 1 guarantees the secrecy of the authenticated message in every protocol session. In the following we say that a process is *nonce linear* iff (*i*) every end assertion has the form $\mathsf{end}^i_n(I,J,M)$ and is preceded by $\mathsf{new}(n)$ and (*ii*) if $\mathsf{end}^i_n(I,J,M)$ occurs inside the scope of a replication then $\mathsf{new}(n)$ occurs inside the scope of the same replication. In fact, this syntactic condition suffices to prove that authenticity on causal graphs implies strong authenticity on the ρ-spi processes abstracted by such causal graphs.

Theorem 4 (Authenticity) *Let P be a process and $\mathscr{C} = graph(P)$ be cycle-invariant. If $\mathscr{C}$ guarantees authenticity, then P guarantees weak authenticity. If $\mathscr{C}$ guarantees authenticity and P is nonce linear, then P guarantees strong authenticity.*

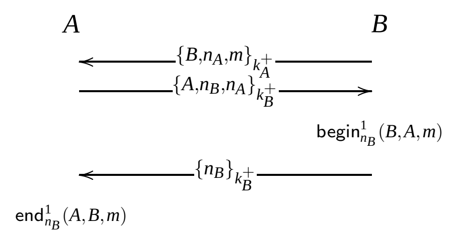
This means that causal graphs, which express the causality among process events, are a sound (and decidable) model for proving weak authenticity while the freshness of authentication requests, namely the condition distinguishing weak from strong authenticity, may be directly verified on the syntax of ρ-spi processes. For example, since *graph*(*Prot*) guarantees authenticity and *Prot* is nonce-linear, then *Prot* guarantees strong authenticity. Notice that *Prot* describes an unbounded number of instances of A acting as claimant in protocol sessions with B and an unbounded number of instances of B acting as verifier in protocol sessions with A. We can easily extend the protocol specification with the parallel composition of A and B running protocol sessions with an arbitrary malicious party E. Even in this scenario, the protocol turns out to be safe.

6 Related Work

Causality-based modelling of concurrency is a widely studied research topic and several important results have been proposed. Event structures [32] are a general and expressive framework for modelling the causality among events in concurrent and distributed systems. This model captures the dependency among events and the interleaving of concurrent events by a partial order. A tricky problem when abstracting away from the multiplicity of protocol sessions is that an event may causally precede itself, thus loosing the antisymmetry property and, consequently, the partial order. This is easily seen by thinking of a process inputting a message and then sending out another message, which is used by the environment to construct a message sent to a replication of the former process and so on. In this scenario, since we abstract away from the multiplicity of protocol sessions, the input is causally preceded by itself. The safety of this kind of abstraction requires to determine which events abstract over the same protocol session and which ones may instead abstract over different protocol sessions. Thus a more specific structure was needed, containing some additional information about message integrity, and relying on paths representing computational flows instead of a partial order among events. Crazzolara and Winskel have applied Petri nets [30], a well-known causality-based model for distributed systems, to the analysis of cryptographic protocols [14], although this work does not abstract away from the multiplicity of sessions.

Type systems proved successful in analyzing different security properties of cryptographic protocols, e.g., [1, 2] for secrecy and [23, 12] for authenticity. As mentioned in the introduction, they exploit syntactic patterns of security protocols while providing guaranteed compositionality, but they constrain the class of analyzable protocols and are typically specific to individual security properties. On the one hand, the type system in [23] is very general and applies to several settings including authorization policies [21] and key-compromise [24], but type definitions quickly get cumbersome and no type-inference algorithm is currently available. The type system in [12] is compositional, modular, and allows for automatic type inference, but this generality is paid by restricting the form of protocols to some specific tagged patterns.

Table 5 Variant of NSL



As a simple example, to the best of our knowledge, the protocol in Table 5 can be analyzed by neither the type system from [23] nor the one from [12]. This protocol is a variation of the Needham-Schroeder-Lowe public-key authentication protocol, in which B sends a message m to A in the first message exchange. From A's point of view, the authentication of this message is guaranteed by the handshake with nonce n_B (second and third message). The problem is that m occurs within neither the second nor the third message, thus resulting free in the corresponding type definitions and not enabling the end assertion based on nonce n_B. Furthermore, as opposed to existing type-systems that in case of failure do not yield any information on possible attacks, a simple inspection of the causal graph gives useful insights on the actual protocol run-time behaviour and, in fact, attack derivation is often immediate.

The control-flow analysis for message authenticity proposed by Bodei *et al.* in [8] and recently extended in [22] to detect replay attacks is closely related to our approach. Although the underlying static analysis techniques are different, both of the approaches rely on an abstraction of the protocol semantics and enjoy guaranteed termination. However, due to the undecidability of secrecy and authenticity, they perform an overapproximation that necessarily rules out safe protocols. Our approach relies on the causality relation among process events while the control-flow analysis of [8] statically verifies the origin and destination of messages and, more precisely, checks whether a message encrypted by A and intended for B does indeed come from A and reaches B only. A formal comparison between the techniques is thus interesting but it is left as future work. Another interesting work is an abstract interpretation for mobile systems and, as an instance, for spi-calculus proposed by Feret [19]. The analysis deals with the origin of messages but does not address the freshness of authentication requests.

Strand spaces [27, 26] are an effective framework for the analysis of cryptographic protocols. Proofs of safety typically rely on the causality among protocol events and, in a recent paper [16], authors investigate how to automatically detect those specific executions, called shapes, among the infinitely many possible, that should be considered for analysis. There are interesting similarities between shapes and causal graphs and, although strand spaces do not enjoy guaranteed termination, we plan to exploit the underlying ideas to further refine the expressiveness of our analysis.

Proverif [5, 3] constitutes a powerful tool that takes as input spi-like protocol descriptions and, by Horn clause resolution, verifies a variety of different security properties such as secrecy, perfect secrecy and authenticity. The analysis is general and automated but guarantees termination only for protocols where every ciphertext is tagged differently [6].

7 Conclusion and Future Work

We have proposed a static analysis technique for analyzing security protocols based on abstract interpretation of the causality among process events. We have specifically shown that secrecy and authenticity can be soundly characterized in terms of causality, but we remark that the analysis is not tailored to these security properties but may as well be applicable to verify properties formulated in terms of causality among the actions of execution traces. The analysis enjoys guaranteed termination since the size of causal graphs is finite, the generation of paths terminates since input edges are inspected only once, thus avoiding loops due to cycles in the graph, and the analysis is linear on the number of paths. We have implemented a tool for automating the analysis, and we have applied the tool to some common protocols in the literature [18]. The analyses terminated within a few seconds and provided safety proofs for the correct versions of the protocols while failing to validate the flawed ones. Remarkably, attacks are often easily derivable by an inspection of the path sets. The only human effort required is to capture the protocol in the dialect of the spi-calculus which is often straightforwardly derivable from the protocol description.

As future work, we plan to investigate a more sophisticated abstraction allowing us to relax some of the constraints that are currently imposed by our analysis: for instance, our experiments show that some false positives occur when an authenticated term, e.g., a session key, has to be kept secret until authentication requests are accepted and is then leaked out. This kind of scenario turns out to be problematic also for the abstraction of Horn clauses in the decidable $\mathcal{H}_1$ subclass proposed by Goubault-Larrecq *et al.* [25]. A first reason for such false positives is that the leakage of names previously kept secret may introduce cycles in the graph that break the cycle-invariance condition. A possible solution is to augment the number of times each input edge can be inspected, by refining function *paths* and the cycle-invariance definition accordingly. This refinement enhances the precision of the abstraction at the price of increasing the number and size of path sets and thus the complexity of the analysis. A second reason is that the check on the freshness of nonces is used so far only as sufficient condition for proving that weak authenticity implies strong authenticity: more generally, nonce checks guarantee that different protocol sessions rely on different nonces and we believe that this information can be used for refining the analysis and, more precisely, for excluding those paths where a check on the same nonce is performed more than once, thus ruling out this kind of false positives.

For the sake of readability, we have not considered operators such as tags and hashes: their insertion in our framework does not induce any complication but is left as future work. Finally, in this paper we have only considered the instantiation of variables with names. The interesting complication arising when variables can be instantiated with ciphertexts is the capability of the environment to forge, and make trusted participants create, arbitrarily nested ciphertexts, thus potentially causing an infinite number of branches in the input. We plan to solve this problem by an overapproximation guaranteeing that the number of ciphertexts generated by trusted principals in the abstract model is finite and by abstracting away from the ciphertexts generated by the environment.

References

[1] M. Abadi. Secrecy by typing in security protocols. *Journal of the ACM*, 46(5):749–786, 1999.

[2] M. Abadi and B. Blanchet. Secrecy types for asymmetric communication. In *Proc. 4th International Conference on Foundations of Software Science and Computation Structures (FOSSACS)*, volume 2030 of *Lecture Notes in Computer Science*, pages 25–41. Springer-Verlag, 2001.

[3] M. Abadi and B. Blanchet. Analyzing security protocols with secrecy types and logic programs. In *Proc. 29th Symposium on Principles of Programming Languages (POPL)*, pages 33–44. ACM Press, 2002.

[4] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *Proc. 4th ACM Conference on Computer and Communications Security*, pages 36–47, 1997.

[5] B. Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *Proc. 14th IEEE Computer Security Foundations Workshop (CSFW)*, pages 82–96. IEEE Computer Society Press, 2001.

[6] B. Blanchet and A. Podelski. Verification of cryptographic protocols: Tagging enforces termination. In *Proc. 6th International Conference on Foundations of Software Science and Computation Structures (FOSSACS)*, pages 136–152, 2003.

[7] D. Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS. In *Advances in Cryptology: CRYPTO '98*, volume 1462 of *Lecture Notes in Computer Science*, pages 1–12. Springer-Verlag, 1998.

[8] C. Bodei, M. Buchholtz, P. Degano, F. Nielson, and H. R. Nielson. Static validation of security protocols. *Journal of Computer Security*, 13(3):347–390, 2005.

[9] M. Boreale. Symbolic trace analysis of cryptographic protocols. In *28rd International Colloquium on Automata, Languages and Programming (ICALP 2001)*, Lecture Notes in Computer Science, pages 667–681. Springer-Verlag, 2001.

[10] M. Boreale, R. D. Nicola, and R. Pugliese. Proof techniques for cryptographic processes. In *Proc. 14th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 157–166, 1999.

[11] M. Bugliesi, R. Focardi, and M. Maffei. Compositional analysis of authentication protocols. In *Proc. 13th European Symposium on Programming (ESOP)*, volume 2986 of *Lecture Notes in Computer Science*, pages 140–154. Springer-Verlag, 2004.

[12] M. Bugliesi, R. Focardi, and M. Maffei. Dynamic types for authentication, 2007. To appear in Journal of Computer Security.

[13] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th Symposium on Principles of Programming Languages (POPL)*, pages 238–252. ACM Press, 1977.

[14] F. Crazzolara and G. Winskel. Events in security protocols. In *Proc. 8th ACM Conference on Computer and Communications Security*, pages 96–105. ACM Press, 2001.

[15] D. E. Denning and G. M. Sacco. Timestamps in key distribution protocols. *Communications of the ACM*, 24(8):533–536, 1981.

[16] S. Doghmi, J. Guttman, and F. Thayer. Searching for shapes in cryptographic protocols. In *Proc. 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Lecture Notes in Computer Science, pages 523–538. Springer-Verlag, 2007.

[17] D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[18] L. S. et Vérification. Security protocols open repository. http://www.lsv.ens-cachan.fr/spore/.

[19] J. Feret. *Analysis of Mobile Systems by Abstract Interpretation*. PhD thesis, École Polytechnique, 2005.

[20] D. Fisher. Millions of .Net Passport accounts put at risk. *eWeek*, May 2003. (Flaw detected by Muhammad Faisal Rauf Danka).

[21] C. Fournet, A. D. Gordon, and S. Maffeis. A type discipline for authorization policies. In *Proc. 14th European Symposium on Programming (ESOP)*, Lecture Notes in Computer Science, pages 141–156. Springer-Verlag, 2005.

[22] H. Gao, P. Degano, C. Bodei, and H. R. Nielson. Detecting replay attacks by freshness annotations. In *WITS '07: Proceedings of the 7th Workshop on Issues in the theory of security*, pages 85–100, 2007.

[23] A. D. Gordon and A. Jeffrey. Types and effects for asymmetric cryptographic protocols. *Journal of Computer Security*, 12(3):435–484, 2004.

[24] A. D. Gordon and A. Jeffrey. Secrecy despite compromise: Types, cryptography, and the pi-calculus. In *Proc. 16th International Conference on Concurrency Theory (CONCUR)*, volume 3653, pages 186–201. Springer-Verlag, 2005.

[25] J. Goubault-Larrecq and F. Parrennes. Cryptographic protocol analysis on real c code. In *Proc. 6th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'05)*, volume 3385 of *Lecture Notes in Computer Science*, pages 363–379. Springer-Verlag, 2005.

[26] J. Guttman, F. Thayer, J. A. Carlson, J. C. Herzog, J. D. Ramsdell, and B. T. Sniffen. Trust management in strand spaces: a rely-guarantee method. In *Proc. 13th European Symposium on Programming (ESOP)*, volume 2986 of *Lecture Notes in Computer Science*, pages 325–339. Springer-Verlag, 2004.

[27] J. D. Guttman and F. J. Thayer. Authentication tests and the structure of bundles. *Theoretical Computer Science*, 283(2):333–380, 2002.

[28] G. Lowe. "A Hierarchy of Authentication Specification". In *Proc. 10th IEEE Computer Security Foundations Workshop (CSFW)*, pages 31–44. IEEE Computer Society Press, 1997.

[29] R. Needham and M. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 12(21):993–999, 1978.

[30] J. L. Peterson. Petri nets. *ACM Computing Surveys*, 9(3):223–252, 1977.

[31] D. Wagner and B. Schneier. Analysis of the SSL 3.0 protocol. In *Proc. 2nd USENIX Workshop on Electronic Commerce*, pages 29–40, 1996.

[32] G. Winskel. Event structures. In *Advances in Petri nets 1986, part II on Petri nets: applications and relationships to other models of concurrency*, pages 325–392. Springer-Verlag, 1987.

[33] T. Y. C. Woo and S. S. Lam. A lesson on authentication protocol design. *Operation Systems Review*, 28(3):24–37, 1994.

A Semantics of ρ-spi

The dynamics of ρ-spi is formalized by means of a transition relation, which is reported in Table 6, between *configurations*, i.e., pairs $\langle s, P\rangle$, where s is a sequence of actions and P is a closed process. Some transitions apply substitutions to processes: formally, a substitution $\sigma : x \mapsto G$ is a function from variables to run-time messages. Often substitutions are written explicitly by $[G_1/x_1, \ldots, G_n/x_n]$. The application of the substitution σ to the process P is denoted by $P\sigma$ and applies only to free occurrences of the variables in P. NAME RES generates a new name n by checking that it differs from all the names already used in the trace s. It is possible to force this condition by applying α-conversion to n, i.e., by substituting n and all of its free occurrences in P with a different name having the same canonical representative. Similar reasoning applies to the restriction of key-pairs. INPUT requires message G, read from the network, to be computable by the environment: the environment knowledge is defined by the message manipulation rules reported in Table 7 and discussed below. The run-time message G is read only if it can be pattern-matched with the input term M via the function *bind*, which is defined in Table 7 and discussed below. We write $\overline{a}$ to denote the decryption key corresponding to a. We have $\overline{n} = n$, $\overline{k_{IJ}} = k_{IJ}$, $\overline{I} = I$, $\overline{k_I^+} = k_I^-$ and $\overline{k_I^-} = k_I^+$. OUTPUT, BEGIN, END, and PRINCIPAL are self-explanatory. Finally, PAR interleaves two different protocol executions and REPLICATION arbitrarily replicates a principal. Function *bind* takes as input a static term M and a run-time message G and, in case it exists, yields the substitution σ which makes M equal to G, up

Table 6 Transition System for ρ-spi

Transition rules: We omit the symmetric rule of PAR.

NAME RES
$$\frac{n \notin \mathrm{bn}(s) \cup \mathrm{fn}(s)}{\langle s, \mathsf{new}(n).P\rangle \rightarrow \langle s :: new(n), P\rangle}$$

KEY-PAIR RES
$$\frac{k_I^-, k_I^+ \notin \mathrm{bn}(s) \cup \mathrm{fn}(s)}{\langle s, \mathsf{new}^{\mp}(k_I).P\rangle \rightarrow \langle s :: new^{\mp}(k_I), P\rangle}$$

INPUT
$$\frac{s \vdash G \qquad \sigma = bind(M,G) \neq \uparrow}{\langle s, \mathsf{in}(M).P\rangle \rightarrow \langle s :: in(M\sigma), P\sigma\rangle}$$

OUTPUT
$$\langle s, \mathsf{out}(G).P\rangle \rightarrow \langle s :: out(G), P\rangle$$

BEGIN
$$\langle s, \mathsf{begin}^i_G(A, I, G').P\rangle \rightarrow \langle s :: begin^i_G(A, I, G'), P\rangle$$

END
$$\langle s, \mathsf{end}^i_G(A, I, G').P\rangle \rightarrow \langle s :: end^i_G(A, I, G'), P\rangle$$

PRINCIPAL
$$\frac{\langle s, P\rangle \rightarrow \langle s :: t, P'\rangle}{\langle s, A \triangleright P\rangle \rightarrow \langle s :: t, A \triangleright P'\rangle}$$

PAR
$$\frac{\langle s, P\rangle \rightarrow \langle s', P'\rangle}{\langle s, P|Q\rangle \rightarrow \langle s', P'|Q\rangle}$$

REPLICATION
$$\langle s, !P\rangle \rightarrow \langle s, P \mid !P\rangle$$

Table 7 Deduction system and binding

Message Manipulation Rules

$$\textsc{Out}\ \frac{out(G)\in s}{s\vdash G}\qquad \textsc{Env}\ \frac{a\notin \mathrm{bn}(s)}{s\vdash a}\qquad \textsc{Pair}\ \frac{s\vdash G_1\qquad s\vdash G_2}{s\vdash (G_1,G_2)}$$

$$\textsc{Pair Des}\ \frac{s\vdash (G_1,G_2)}{s\vdash G_1\qquad s\vdash G_2}\qquad \textsc{Enc}\ \frac{s\vdash G\qquad s\vdash a}{s\vdash \{G\}_a}\qquad \textsc{Dec}\ \frac{s\vdash \{G\}_a\qquad s\vdash \overline{a}}{s\vdash G}$$

$$\textsc{Public Keys}\ \ s\vdash k_I^+\qquad \textsc{Enemy Keys}\ \ s\vdash k_{EI}\qquad s\vdash k_{IE}\qquad s\vdash k_E^-$$

Binding

$$\begin{array}{ll}
bind(a,a)=[] & \\
bind(x,a)=[a/x] & \\
bind((M,M'),(G,G'))=bind(M,G)\uplus bind(M',G') & \\
bind(\{M\}_{\overline{a}},\{G\}_a)=bind(M,G) & \\
bind(M,G)=\uparrow & \text{otherwise}
\end{array}$$

Table 8 Deduction System and Binding

$$\textsc{Out}\ \frac{\mathsf{G}\in\mathcal{G}}{\mathcal{G}\vdash\mathsf{G}}\qquad \textsc{Env}\ \ \mathcal{G}\vdash\mathscr{E}\qquad \textsc{Pair}\ \frac{\mathcal{G}\vdash\mathsf{G}_1\qquad \mathcal{G}\vdash\mathsf{G}_2}{\mathcal{G}\vdash(\mathsf{G}_1,\mathsf{G}_2)}$$

$$\textsc{Pair Des}\ \frac{\mathcal{G}\vdash(\mathsf{G}_1,\mathsf{G}_2)}{\mathcal{G}\vdash\mathsf{G}_1\qquad \mathcal{G}\vdash\mathsf{G}_2}\qquad \textsc{Enc}\ \frac{\mathcal{G}\vdash\mathsf{G}\qquad \mathcal{G}\vdash\mathsf{v}}{\mathcal{G}\vdash\{\mathsf{G}\}_{\mathsf{v}}}$$

$$\textsc{Dec}\ \frac{\mathcal{G}\vdash\{\mathsf{G}\}_{\mathsf{v}}\qquad \mathcal{G}\vdash\mathsf{v}'\qquad \mathsf{inv}(\mathsf{v},\mathsf{v}')}{\mathcal{G}\vdash\mathsf{G}}$$

Binding

$$\begin{array}{ll}
\mathsf{bind}(\mathsf{v}_1,\mathsf{v}_2)=(\mathsf{v}_2,[]) & \text{if } [\mathsf{v}_1]=[\mathsf{v}_2]\ \vee \\
 & \quad([\mathsf{v}_1]=\mathscr{E}\wedge[\mathsf{v}_2]\in PN)\ \vee \\
 & \quad([\mathsf{v}_2]=\mathscr{E}\wedge[\mathsf{v}_1]\in PN) \\
\mathsf{bind}(\mathsf{x},\mathsf{v})=(\mathsf{v}_{(\mathsf{x})},[\mathsf{v}_{(\mathsf{x})}/\mathsf{x}]) & \\
\mathsf{bind}((\mathsf{M},\mathsf{M}'),(\mathsf{G},\mathsf{G}'))= & \text{if } \mathsf{bind}(\mathsf{M},\mathsf{G})=(\mathsf{G}_1,\sigma_1)\ \wedge \\
\quad((\mathsf{G}_1,\mathsf{G}_2),(\sigma_1\uplus\sigma_2)) & \quad \mathsf{bind}(\mathsf{M}',\mathsf{G}')=(\mathsf{G}_2,\sigma_2) \\
\mathsf{bind}(\{\mathsf{M}\}_{\mathsf{v}},\{\mathsf{G}\}_{\mathsf{v}'})=(\{\mathsf{G}'\}_{\mathsf{v}'},\sigma) & \text{if } \mathsf{bind}(\mathsf{M},\mathsf{G})=(\mathsf{G}',\sigma)\ \wedge \\
 & \quad \mathsf{inv}(\mathsf{v},\mathsf{v}') \\
\mathsf{bind}(\mathsf{M},\mathsf{G})=\uparrow & \text{otherwise}
\end{array}$$

Notation:

$PN\triangleq\{\mathsf{a}\mid\exists\mathsf{k},\mathsf{I},\mathsf{E}\text{ s.t. }\mathsf{a}\in\{\mathsf{I},\mathsf{k}_\mathsf{I}^+,\mathsf{k}_\mathsf{E}^-,\mathsf{k}_{\mathsf{IE}},\mathsf{k}_{\mathsf{EI}},\mathscr{E}\}\}$.
In Out, Pair Des, and Dec, G, G_1, and G_2 are not in *PN*.

$$\begin{array}{ll}
\mathsf{inv}(\mathsf{v},\mathsf{v}')\triangleq & ([\mathsf{v}']\neq\mathscr{E}\Rightarrow\mathsf{bind}(\mathsf{v},\overline{[\mathsf{v}']})\neq\uparrow)\ \wedge \\
 & ([\mathsf{v}]\neq\mathscr{E}\Rightarrow\mathsf{bind}(\overline{[\mathsf{v}]},\mathsf{v}')\neq\uparrow)
\end{array}$$

to the different notation for encryption and decryption keys. If pattern-matching fails, *bind* returns $\uparrow$. This function is defined by cases on the structure of term M: a name matches a name with empty substitution; a variable can be bound to a name; pairs match pairs yielding a substitution which is the union $\uplus$ of the ones achieved for the subterms; finally, decryptions must be performed with the correct decryption key. In all the other cases, *bind* returns failure $\uparrow$.

The knowledge of the environment is formalized by the deduction system reported in Table 7. Rule Out says that every message sent on the network is known by the environment. Env allows the environment to know any name which is not bound (i.e., restricted) in the trace. By Pair and Pair Des, the environment can construct and destruct pairs. By Enc, and Dec the environment can encrypt and decrypt messages only knowing the required keys. By Public Keys, all the public keys are known to the environment. Finally, by Enemy Keys, the environment may be provided with its own private keys and with long-term keys shared with honest participants. This gives the possibility to the enemy to interact with the other participants by pretending to be a trusted principal.

B Semantics of Causal Graphs

The knowledge of the abstract environment is formalized by the judgement $\mathcal{G}\vdash\mathsf{G}$, meaning that the environment can construct G given the knowledge of the terms in $\mathcal{G}$: this judgement is defined by a deduction system reported in Table 8. The environment knows every term sent on the network (Out) and the special name $\mathscr{E}$ (Env), it can construct and destruct pairs (Pair and Pair Des) and encrypt and decrypt terms only knowing the required keys (Enc and Dec). For reducing the number of terms known to the environment and abstracting the same set of ρ-spi terms, we prevent the environment from deriving labelled version of identities, public keys, enemy's keys and $\mathscr{E}$, which abstract the same ρ-spi terms as $\mathscr{E}$.

Function bind, reported in Table 8, defines the pattern-matching among terms. This function takes as input a term M and a ground term G and, if the two terms match, yields the term G' obtained by labelling G according to the variables in M and the corresponding substitution. If pattern-matching fails, bind returns $\uparrow$. Function bind is defined by cases on the structure of term M: a name matches itself and, similarly, the name $\mathscr{E}$ matches identities, public keys and enemy's keys; a variable can only be bound to an atomic name v; pattern-matching of pairs and ciphertexts is defined component-wise: notice that decryptions must be performed by the correct decryption key. In all the other cases, bind returns failure $\uparrow$. Function $\overline{\mathsf{v}}$ yields the decryption key matching the encryption key v: this function is smoothly extended to arbitrary abstract terms G by inverting the encryption keys in G.