Report

Integrated Specification and Verification of Security Protocols and Policies

Author(s):
Frau, Simone; Torabi Dashti, Mohammad

Publication Date:
2010

Permanent Link:
https://doi.org/10.3929/ethz-a-006935684

Rights / License:
In Copyright - Non-Commercial Use Permitted

This page was generated automatically upon download from the ETH Zurich Research Collection. For more information please consult the Terms of use.
Integrated Specification and Verification of Security Protocols and Policies

Simone Frau  
ETH Zürich

Mohammad Torabi-Dashti  
ETH Zürich

ABSTRACT
We propose a language for formal specification of service-oriented architectures. The language supports the integrated specification of communication level events, policy level decisions, and the interaction between the two. We show that the reachability problem is decidable for a fragment of service-oriented architectures. The decidable fragment is well suited for specifying, and reasoning about, security-sensitive architectures. In the decidable fragment, the attacker controls the communication media. The policies of services are centered around the trust application and trust delegation rules, and can also express RBAC systems with role hierarchy. The fragment is of immediate practical relevance: we have specified and verified a number of security-sensitive architectures stemming from the e-government domain.

1. INTRODUCTION
Security protocols and authorization logics are two major techniques used in securing software systems. A central role of any (security) protocol is to give meaning to the messages that are exchanged in the course of the protocol [16]. For example, a signed X.509 certificate sent by a certificate authority in many security protocols meant to imply that the authority endorses the public key and its owner, mentioned in the certificate. There are several ways to make the meanings of messages, and in general actions, of a protocol explicit, e.g. by associating epistemic effects to the actions [8]. In this paper, we propose a formal language for specifying service-oriented architectures, in which

- the messages received by a service are interpreted in terms of policy statements of the service, and
- the authorization policies of the service constrain the actions the service can perform.

The proposed language is well suited for integrated specification of security protocols and authorization policies in service-oriented architectures. We see a service-oriented architecture as a collection of finitely many services which communicate over insecure media. Each service consists of a number of processes that run in parallel and share a policy engine. Processes communicate by sending and receiving messages, as it is usual in asynchronous message passing environments. Each send event is constrained by a guard, and each receive event leads to an update. Guards and updates belong to the policy level, as opposed to send and receive events which constitute the communication level. In anthropomorphic terms, services “think” at the policy level, and “talk” at the communication level.

From an operational point of view, guards are predicates which, if derivable by the policy engine of a service, allow the service to perform a corresponding send action, cf. Dijkstra’s guarded command language. Updates are also predicates at the policy level. When a service receives a message in one of its processes, it adds the corresponding update predicates to its policy engine. Intuitively, updates associate meanings to the messages a service receives in terms of predicates in the policy level. The notion of updates is similar to the assumptions which are relied upon after receiving a message, in the trust management model of Guttman et al. [13].

Motivations. The separation between the communication and policy levels is a useful abstraction for better understanding each of these levels. Indeed, distributed authorization logics, such as [7, 4, 11], typically abstract away the communication level events by assuming that all the policy statements exchanged among the participants are signed certificates. This frees the modelers from specifying the exact routes through which the statements travel, etc. The abstraction however obscures how each of the policy statements are represented (as messages) in a given application, how messages are interpreted as policy statements, whether there is a place for misinterpretation, etc. For instance, one would expect that the statement “Ann says employee(Piet)” is added to the (e.g. Binder [7]) policy engine of a service, only after the service receives a message which is meant to indicate that Piet is Ann’s employee. However, the meanings of messages (determined by their format, who has signed them, etc.) is often not specified in the policy level. Therefore, in a concrete environment, it is unclear whether or not the attacker can fake a message which would mean that Piet is Ann’s employee, even though he is not. Similarly, formal specifications of security protocols, e.g. as in [15, 17], fully detail the format of the exchanged messages, while the meanings of messages in terms of policy statements are left unspecified.

While maintaining the separation between the communica-
tion and policy levels, we believe that, for a thorough security analysis, the interaction between the two levels must also be defined precisely. A typical specification in our proposed language thus consists of three components: communication level events, policy level decisions, and the interface between the two. As the interface between the levels is explicitly present in the specifications, a more precise security analysis of service-oriented architectures becomes possible. This singles out our specification language from the formalisms which focus on either the communication or the policy level, and hence neglect their interactions.

**Decision procedure.** We assume that the attacker is in direct control of the communication media, i.e. messages are passed through the attacker. The message composition capabilities of the attacker may, for example, reflect the Dolev-Yao threat model [9]. The attacker can indirectly affect the policies of the participating services, by sending tampered messages which in turn affect the update predicates.

A generic reachability problem is defined for service-oriented architectures specified in the language. The reachability problem subsumes the secrecy problem for security protocols and the safety problem for authorization policies. The reachability problem turns out to be undecidable in general, even when assuming a finite bound on the number of participating services. We give a decision algorithm for the reachability problem under the following two conditions: (1) the message composition and decomposition capabilities of the attacker reflect the Dolev-Yao threat model, and (2) policy engines of (honest) services are centered around the trust application and trust delegation rules à la dkal [11], besides type-1 theories (formally defined in section 4). Type-1 theories are sufficiently expressive for modelling, e.g., RBAC systems with role hierarchy. The trust application and trust delegation rules, which are the core of many distributed authorization logics [7, 4, 11], intuitively state that

**Trust application** If Ann trusts Mike on statement \( f \), and Mike says \( f \), then Ann believes \( f \) holds.

**Trust delegation** If Ann trusts Mike on statement \( f \), and Mike delegates the right to state \( f \) to, e.g., Piet, then Ann trusts Piet on statement \( f \).

Trust delegation often contributes to the resilience and flexibility of access control systems. In practice, however, for a given application, trust delegation may, or may not, be allowed. Our formalism and decision algorithm can be adapted to exclude (transitive) trust delegation, if desired.

The decidable fragment is of practical interest: several industrial service-oriented architectures studied in the context of AVANtSSAR [2] (The EU Project on Automated Validation of Trust and Security of Service-oriented Architectures) fall into this fragment. As a comprehensive example, a case study on specifying and verifying an on-line car registration service [2], stemming from the European initiative for points of single contact, is reported in this paper.

To prove our decidability result, we encode the derivation of authorization predicates in the policy engine of a service into message inference trees induced by the Dolev-Yao deduction rules. The encoding benefits us in two ways: (1) it simplifies the decidability proof, and (2) it allows us to build upon existing tools which have been originally developed for verifying security protocols. In particular, we have extended the constraint solver of Mälen and Shimatikov [15] in Prolog to validate service-oriented architectures.

Note that verification algorithms for correctness of security protocols and authorization logics have been mostly developed in isolation. For instance, it has been shown that the secrecy problem is decidable for security protocols with a bounded number of sessions [17, 15]. For these results the local computational power of the processes is limited to pattern matching, hence not fully accounting for authorization policies of the participants. Likewise, (un)decidability results for the safety problem in the HRU access control matrix model, and authorization logics such as [7, 4, 11] abstract away communication level events and their effects on policy level decisions. In contrast, our decision algorithm for reachability takes the communication and policy levels into account, and also covers the interface between them.

**Related work.** Our proposed language can be used to specify security protocols as it is common in the literature, e.g. see [15]. Authorization policies are modeled as logic programs in the language. Logic programs have been extensively used for specifying and reasoning about policies, e.g. see Binder [7], secpal [4], and dkal [11].

Recent progress in analyzing business processes, augmented with authorization policies, is related to our work, e.g. see [1, 18]. These studies focus on using specific formalisms and techniques for selected case studies, and thus do not consider decidability issues in general. A notable exception to this is [3], where workflows, policy level predicates and their interfaces are all formalized in first-order logic. Reachability is not considered in [3].

**Road map.** Section 2 describes an on-line car registration service: this serves as our running example. Section 3 introduces the syntax and semantics of the language we propose for specifying service-oriented architectures. There, we also formally define the reachability problem. A decidable fragment of architectures is identified in section 4. A constraint-solving algorithm for deciding reachability in the fragment is given in section 5. Section 6 concludes the paper. The proofs that are omitted in the text can be found in appendix B.

## 2. A RUNNING EXAMPLE: CRP

We give an informal description of a service-oriented architecture for online car registration procedure, originated from the European initiative for points of single contact, see [2]. We refer to the case study as CRP. A formal model of CRP and its verification results are given in subsequent sections.

CRP involves a number of parties: Mike the new owner of a car, Piet the employee of the car registration office, Ann the head of the car registration office, the human resources department of the office, called hr, and the central repository server, referred to as cr.
Mike buys a car, but before he is allowed to drive it, he must register the car at the car registration office. The office provides an online registration service. After producing a document containing all the necessary data for registering the car, Mike sends the document to one of the employees of the office, Piet. If the document is valid, Piet sends it to the central repository server to be permanently stored. The cr allows only the employees of the car registration office to write on the server, and Piet is not initially known to the cr as an employee. The cr trusts Ann, the head of the office, on who is an employee of the office. Ann, however, has decided to delegate this task (or, right) to the human resources department hr, and communicates this decision to the cr with a certificate. Consequently, the cr inquires the hr on the status of Piet, to which the hr replies with a certificate stating that Piet is an employee of the office. Finally, the cr accepts Piet’s request to store the document. After storing the document, the cr sends back an acknowledgement to Piet. Piet, in turn, sends a token of successful completion of the registration to Mike.

Below, we present the message exchange pattern for the CRP. The usual primitives for security protocols are employed: asymmetric encryptions \{\}, signatures \text{sig}(\cdot, \cdot), public key constructors \text{pk}(\cdot), hash functions \text{h}(\cdot) and pairing \langle\cdot, \cdot\rangle. When confusion is unlikely, we simply write \text{x, y} for \text{a, x, y, sig(a, x)}. We assume these cryptographic operators are ideal, à la Dolev and Yao [9]. Below, terms in sans-serif are flags, i.e. unique constants which denote the purpose of their accompanying messages.

\[
\begin{align*}
\text{mike} &\rightarrow \text{piet} : \{\text{mike, doc}\}_{\text{pk(piet)}}, [\text{h(doc)}]_{\text{mike}} \\
\text{piet} &\rightarrow \text{cr} : \{\text{mike, doc}\}_{\text{pk(cr)}}, [\text{ann, h(mike, doc)}]_{\text{piet}} \\
\text{cr} &\rightarrow \text{piet} : [\text{h(mike, doc)}]_{\text{cr}} \\
\text{piet} &\rightarrow \text{mike} : [\text{h(mike, doc)}, \text{success_token}]_{\text{piet}}
\end{align*}
\]

The delegation of Ann’s right to the hr, and the hr’s attestation that Piet is an employee, go over a separate exchange.

\[
\begin{align*}
\text{cr} &\rightarrow \text{ann} : \text{piet, empti_status} \\
\text{ann} &\rightarrow \text{cr} : [\text{piet, is, empti, delegated_to, hr}]_{\text{ann}} \\
\text{cr} &\rightarrow \text{hr} : \text{piet, empti_status} \\
\text{hr} &\rightarrow \text{cr} : [\text{piet, is, empti}]_{\text{hr}}
\end{align*}
\]

This specification falls short of capturing the internal reasonings of the participants involved, as described informally above. For instance, it is not clear how the participants must interpret the messages they receive, and how the cr ascertains Piet’s right to store. These relations are often formalized in authorization logics, such as DKAL. However, specifying the internal reasoning in, e.g. SECPAL and DKAL, would fall short of determining the actual messages exchanged, or how the messages are related to policy statements.

In section 4.1, we specify CRP in our formal language. Then, we can answer questions such as: whether the attacker can take the CRP system into a state in which, Eve, who is not an employee of the office, can write into the repository. Such questions cannot be meaningfully posed when considering the communication or the policy levels of CRP in isolation.

3. A LANGUAGE FOR SPECIFYING SERVICE ORIENTED ARCHITECTURES

The syntax and semantics of the language are defined below. A typical architecture specified in the language consists of communication events of services, how received messages are interpreted, how services make logical decisions, and how these decisions affect the communication events.

3.1 Syntax

A signature is a tuple \((\Sigma, \mathcal{V}, \mathcal{P})\), where \(\Sigma\) is a countable set of functions, \(\mathcal{V}\) is a countable set of variables, \(\mathcal{P}\) is a nonempty finite set of predicates, and these sets are pairwise disjoint. We use the capital letters \(A, B, \ldots\) to denote the elements of \(\mathcal{V}\). The free term algebra induced by \(\Sigma\), with variables \(\mathcal{V}\), is denoted \(T_\Sigma(\mathcal{V})\). A message is an element of \(T_\Sigma(\mathcal{V})\), i.e. a ground term. The set of atoms \(A_{0}(\mathcal{V})\) is defined as \(\{p(t_1, \ldots, t_n) \mid p \in \mathcal{P}, t_i \in T_\Sigma(\mathcal{V}), \text{arity of } p = n\}\). The total function \(\text{var} : T_\Sigma(\mathcal{V}) \cup A_{0}(\mathcal{V}) \rightarrow 2^\mathcal{V}\) gives the set of variables appearing in terms and atoms. A fact is an atom with no variables.

Fix a signature. An event is of either of the following forms:

- \(g_i \vee \cdots \vee g_k \triangleright s(t)\),
- \(r(t) \triangleright u\)

where \(t \in T_\Sigma(\mathcal{V})\), each \(g_i\), with \(1 \leq i \leq k\) and \(k \geq 0\), is a finite set of atoms, and \(u\) is a finite set of atoms. Intuitively, an event of the form \(g_i \vee \cdots \vee g_k \triangleright s(t)\) denotes \text{guarded send}, where term \(t\) is sent to the network only if the guard \(g_i \vee \cdots \vee g_k\) is evaluated to “true” (for the exact definition, see section 3.2). An event of the form \(r(t) \triangleright u\) denotes \text{receive followed by update}, where receiving term \(t\) results in adding the update \(u\) to the policy level. The function \(\text{var}\) extends to events as \(\text{var}(g_i \vee \cdots \vee g_k \triangleright s(t)) = \bigcup_{1 \leq i \leq k} \text{var}(g_i) \cup \text{var}(t)\) and \(\text{var}(r(t) \triangleright u) = \text{var}(t) \cup \text{var}(u)\).

We write \(E\) for the set of all events, and \(E^+\) for the set of all finite sequences of events. A process is a finite sequence of events \(e_1 \cdots e_n\) where

- If \(e_i = g_1 \vee \cdots \vee g_k \triangleright s(t)\), then \(\text{var}(e_i) \subseteq \bigcup_{1 \leq j \leq i} \text{var}(e_j)\), for all \(1 \leq i \leq n\).
- If \(e_i = r(t) \triangleright u\), then \(\text{var}(u) \subseteq \text{var}(t) \bigcup_{1 \leq j \leq i} \text{var}(e_j)\), for all \(1 \leq i \leq n\).

These conditions intuitively state that the behavior of any process depends deterministically on its input.

A service \(\pi\) is a tuple \((\eta, \Omega, \var_0)\), where \(\eta\) is a finite set of processes, \(\Omega\) is a finite set of facts, called the knowledge of \(\pi\), and \(\var_0\) is a finite set of Horn clauses, called the \text{intensional knowledge of } \pi. A Horn clause is of the form \(a \leftarrow a_1, \cdots, a_n\), with \(n \geq 0\), and \(a, a_1, \cdots, a_n\) being atoms.

An attacker model \(A\) is a service with no processes, i.e. it is a pair \((\Omega, \var_0)\), where \(\Omega\) is a finite set of facts, called the knowledge of the attacker, and \(\var_0\) is a finite set of Horn clauses, referred to as the \text{intensional knowledge of the attacker}. The attacker is also able to send and receive messages; these capabilities are reflected in the execution model described in section 3.2.
A (service-oriented) architecture is a tuple \((\Sigma, V, P), \Pi, A)\), where \((\Sigma, V, P)\) is a signature, \(\Pi\) is a finite nonempty set of services and \(A\) is an attacker model, where \(\Pi\) and \(A\) are defined using the signature \((\Sigma, V, P)\). In order to avoid trivial name clashes, it is assumed that variables that appear in different processes of an architecture are distinct.

We assume that for any architecture \((\Sigma, V, P), \Pi, A)\), the predicate \(K\), of arity 1, belongs to \(P\). This predicate is in particular used to model the knowledge of the attacker \(A\).

### 3.2 Semantics

Let \(s\) be a finite set of facts, and \(I\) be a finite set of Horn clauses. The closure of \(s\) under \(I\), denoted \([s]^{I}\), is the smallest set that contains \(s\) and moreover \(\forall a \leftarrow a_{1}, \ldots, a_{n} \in I \exists a \sigma \in [s]^{I} \Rightarrow a \sigma \in [s]^{I}\), where \(a\) is a total (grounding) substitution function for the Horn clause \(a \leftarrow a_{1}, \ldots, a_{n}\); that is \(\sigma : \langle \text{var}(a) \cup \{a_{1}, \ldots, a_{n}\} \rangle \rightarrow T\).

Let \((\Sigma, V, P), \Pi, A)\) be an architecture, consisting of services \(\Pi = \{1, \ldots, \ell\}\), a configuration \(I\) of the architecture is a tuple \((\langle \eta_{1}, \Omega_{1}\rangle, \ldots, \langle \eta_{\ell}, \Omega_{\ell}\rangle, \Omega_{A}\)) where \(\eta_{i}\) is a finite set of processes, for \(1 \leq i \leq \ell\), and \(\Omega_{A}\) and \(\Omega_{\ell}\) are finite sets of facts. The initial configuration of the architecture is \(z^{0} = \langle\langle \eta_{1}, \Omega_{1}\rangle, \ldots, \langle \eta_{\ell}, \Omega_{\ell}\rangle, \Omega_{A}\rangle\).

Each architecture \((\Sigma, V, P), \Pi, A)\) is attributed with a Kripke structure which represents all the executions of the architecture. This is explained informally in the following. Suppose a process belonging to service \(\pi \in \Pi\) can perform a guarded send event \(g_{1} \lor \cdots \lor g_{k} \blacktriangleright s(t)\). Then, the guard is evaluated against the policy engine of \(\pi\). The guard is interpreted as the “disjunction” of the “conjunctions” of atoms in each \(g_{i}\), with \(1 \leq i \leq k\). If the guard can be derived in the policy engine (i.e. the guard evaluates to “true”), then the process sends the term \(t\) to the network; that is, \(t\) is immediately added to the knowledge of the attacker. We remark that variables appearing in a guarded send event originate in previous receive events in the process. Therefore, the variables in term \(t\) have already been instantiated with ground values. Therefore, only messages (i.e. terms with no variables) are sent to the network. Now, suppose a process belonging to service \(\pi \in \Pi\) can perform a receive event \(r(t) \triangleright u\). If there exists a grounding substitution \(\sigma\) such that \(r \sigma\) can be derived from the attacker’s knowledge, then the process receives \(r \sigma\) and the predicates \(\sigma r\) are added to the knowledge set of service \(\pi\). The formal definition is given below.

An architecture \((\Sigma, V, P), \Pi, A)\), with \(\Pi = \{1, \ldots, \ell\}\), induces a Kripke structure \((S, S^{0}, T)\), where \(S\) and \(T\) are the smallest sets satisfying

- \(S^{0} = z^{0}, z^{0} \in S\).
- If \(z = \langle\langle \eta_{1}, \Omega_{1}\rangle, \ldots, \langle \eta_{\ell}, \Omega_{\ell}\rangle, \ldots, \langle \eta_{\ell}, \Omega_{\ell}\rangle, \Omega_{A}\rangle \in S\), then
  - If \(e \in \eta_{i}\) with \(e = \langle g_{1} \lor \cdots \lor g_{k} \blacktriangleright s(t)\rangle, e' \in E^{*}\), and \(\exists j, 1 \leq j \leq k \land g_{j} \subseteq [\Omega_{i}]^{1}\), then \(z' \in S\) and

Given a configuration \(Z\) and Kripke structure \((S, S^{0}, T)\), we say \(Z\) is reachable in \((S, S^{0}, T)\) iff \((S^{0}, Z) \in T^{*}\), where \(T^{*}\) is the reflexive transitive closure of \(T\).

### 3.3 The reachability decision problem

Given an architecture \(arch = (\Sigma, V, P), \Pi, A)\), with \(\Pi = \{1, \ldots, \ell\}\) and a fact \(f \in \mathcal{A}_{\Sigma(0)}\), the reachability problem \(\text{Reach}(arch, a, f)\), with \(a \in \Pi \cup \{A\}\), asks whether there exists a reachable configuration \(\langle\langle \eta_{1}, \Omega_{1}\rangle, \ldots, \langle \eta_{\ell}, \Omega_{\ell}\rangle, \Omega_{A}\rangle\) in the Kripke structure induced by the architecture such that \(f \in [\Omega_{A}]^{1}\), or not. These cases are respectively denoted by \(\text{Reach}(arch, a, f) = T\) and \(\text{Reach}(arch, a, f) = F\).

The decision problem \(\text{Reach}\) subsumes the secrecy problem for security protocols and the safety problem for authorization logics. The secrecy problem asks whether the attacker can learn a (supposedly secret) message \(m\) via interacting with other services. This can be represented as \(\text{Reach}(arch, A, K(m))\). The safety problem asks whether an authorization predicate \(f\) (e.g. \(\text{knows(can\text{-}read(ann, file12))}\)) can be derived by a service, say \(\pi\), with \(\pi \in \Pi\). This corresponds to \(\text{Reach}(arch, \pi, f)\).

Due to the computational power of logic programs, \(\text{Reach}\) is in general undecidable (even though any architecture consists of finitely many services). It is easy to observe that \(\text{Reach}\) is semi-decidable if the ground deduction problem is decidable for the intensional knowledge sets of the participants and the attacker. The ground deduction problem for a finite set of Horn clauses \(I\), asks whether \(a \in \{s\}^{1}\), for an arbitrary fact \(a\) and a finite set of facts \(s\).

### 4. A DECIDABLE FRAGMENT

In this section, we identify a fragment of intensional knowledge (for the participants and the attacker) that admits decision algorithms for the reachability problem. In this decidable fragment, referred to as \(A_{1}\), the intensional knowledge of the attacker is fixed to the standard Dolev-Yao (DY) deduction rules, as formalized in, e.g., [15]. The policies (i.e. intensional knowledge) of honest services are centered around trust application \(TA\), and (transitive) trust delegation \(TD\) rules, adopted and adapted from \(DKAL\), and can also express typical RBAC systems with role hierarchy. Next, we introduce the notion of infons, cf. [11, 12]. Infons are pieces of information, e.g. \(\text{can\text{-}read(piet, file12)}\) stipulating that Piet can read a certain file12. An infon does not admit a truth value, i.e. it is never false or true. Instead, infons are the interfaces between the communication level and policy level for honest services. That is, if the policy engine of a service, say \(Ann\), derives the predicate \(\text{knows(can\text{-}read(piet, file12))}\), then \(Ann\) “knows” that Piet may read this file, and may thus grant him read access to \(file12\). Note that “knows” in this context, and also in \(DKAL\),
is a predicate symbol and not a modality as in logics of knowledge. In fact, “knows” here is closer to the notion of belief rather than knowledge, in epistemic terms.

Infons are different from predicates (i.e., policy statements) in that they can be nested, i.e., infons are constructed by applying infon constructors to message terms and other infons. We assume that in any signature \((\Sigma, V, P)\), the set \(\Sigma\) can be partitioned into \(\Sigma_{\text{msg}}\) and \(\Sigma_{\text{infon}}\), so that \(\Sigma_{\text{infon}}\) is the set of infon constructor functions, and \(\Sigma_{\text{msg}}\) is the set of message constructor functions. The set of infons \(\text{Infons}\) is formally defined as the smallest set satisfying the property: if \(t_1, \ldots, t_n \in T_{\Sigma_{\text{msg}}(V)} \cup \text{Infons}\) and \(f \in \Sigma_{\text{infon}}\) with arity \(f\) being \(n\), then \(f(t_1, \ldots, t_n) \in \text{Infons}\). Note that \(\text{Infons} \cap T_{\Sigma_{\text{msg}}(V)} = \emptyset\); in particular messages are not infons.

We assume that for honest services the policy statements (i.e. the inhabitants of the policy level) are predicates over \(\text{Infons}\). That is, for these services, the knowledge ranges over pieces of information. In contrast, the knowledge of the attacker ranges directly over message terms. Below, disjoint union of two sets is denoted by \(\sqcup\).

**Definition 1.** Fragment \(A_1\) consists of all architectures 

\[
\text{arch} = ((\Sigma, V, P), \Pi, A),
\]

which satisfy the following syntactical conditions:

- \((\Sigma, V, P)\) is a signature, with \(\Sigma = \Sigma_{\text{msg}} \cup \Sigma_{\text{infon}}\), and:
  - A finite subset of constants in \(\Sigma_{\text{msg}}\), denoted by Agents, represents the set of the names of the services in \(\Pi\).
  - Apart from nullary functions (i.e. constants), \(\Sigma_{\text{msg}}\) only contains the functions \(\{\cdot\}, \lfloor \cdot \rfloor, \lceil \cdot \rceil, \text{sig}(\cdot), \text{pk}(\cdot), \text{h}(\cdot), (\cdot), (\cdot)\). These represent respectively asymmetric and symmetric encryption, digital signature, public key constructor, hash and pairing functions, interpreted as usual.
  - \(\Sigma_{\text{infon}}\) contains in particular the functions \(\theta(\cdot, \cdot)\) and \(\sigma(\cdot, \cdot)\). These are intuitively stand for trusted on and said, respectively, with \(\theta, \sigma : \text{Agents} \times \text{Infons} \rightarrow \text{Infons}\).
  - \(\Pi = \{K\}\), with \(K\) being an unary predicate. Intuitively, \(K\) stands for “knows”.

- A service \(\pi = (\eta_i, \Omega_i, I_i)\) in \(\Pi\) meets the conditions:
  - For all processes in \(\eta_i\), all the terms sent and received are elements of \(T_{\Sigma_{\text{msg}}(V)}\).
  - \(\Omega_i\) is a finite set of ground atoms of the form \(K(i)\), with \(i \in \text{Infons}\).
  - \(I_i\) includes the TA and TD rules, respectively represented by \(K(X) \leftarrow K(\theta(A, X)), K(\sigma(A, X))\), and \(K(\theta(A, \theta(B, X))) \leftarrow K(\theta(A, X))\). The set of all the other rules in \(I_i\) constitutes a type-1 theory, as defined below, and \(\sigma\) and \(\theta\) do not appear in this set.

\(^2\)Intuitively, one (or more) public key is attributed to each element of Agents. The public keys are known to everyone, and to the attacker in particular. Using the public key of a \(\in\) Agents one can encrypt messages for a and can verify the authenticity of the messages signed by a.

**Figure 1:** Dependency graph, example 1.

**Example 1.** Consider a file server which implements an RBAC system with two roles, user and admin. Users may read any public file, admins may read any classified file, and admins may also write to any file. Admins inherit all the rights attributed to users. Below, we give a type-1 theory which describes this RBAC system.

\[
\begin{align*}
K(\text{user}(A)) & \leftarrow K(\text{admin}(A)) \\
K(\text{can_read}(A, F)) & \leftarrow K(\text{user}(A), K(\text{public}(F))) \\
K(\text{can_read}(A, F)) & \leftarrow K(\text{admin}(A), K(\text{classified}(F))) \\
K(\text{can_write}(A, F)) & \leftarrow K(\text{admin}(A))
\end{align*}
\]

Here \(\Sigma_{\text{msg}}\) contains the set of identities of involved services, and names of files, while \(\Pi\) is \(\{K\}\), and \(\Sigma_{\text{infon}} = \{\text{user}, \text{admin}, \text{public, classified, can_read, can_write}\}\) with obvious arities. The infon dependency graph for this theory, shown in figure 1, is indeed acyclic.

The type-1 policy of examples 1 is centralized in the sense that it describes the policies of a single entity, i.e., the file server. In the context of an architecture, this type-1 theory may represent the policies of a single service, cf. section 4.1.
In section 5, we give a decision algorithm for the reachability problem for architectures in fragment $\mathbb{A}_1$. The decision algorithm is based upon encoding policy level computations of services into message derivation trees of the standard DY model. Let us now conclude with a formal specification of CRP (section 2) as an $\mathbb{A}_1$ architecture.

4.1 A formal specification of CRP

We give a formal specification of CRP by defining the architecture $crp = ((\Sigma, V, P), \Pi, A)$. The architecture, as we will see, indeed falls into the $\mathbb{A}_1$ fragment. The signature $\Sigma$ contains the standard cryptographic primitives $\{-\cdot\}, \{\cdot\cdot\}, pk(\cdot), h(\cdot), sig(\cdot, \cdot), (\cdot, \cdot) \in \Sigma_{msg}$, and the infor constructors, namely can_store(\cdot), emp\l(\cdot), head(\cdot), $\sigma(\cdot, \cdot), \theta(\cdot, \cdot) \in \Sigma_{infor}$.

The set of constants $\Sigma_C \subset \Sigma_{msg}$ used in $crp$ is defined as $\Sigma_C = \{ \text{mike, piet, ann, cr, hr, eve, doc, emp\l status, is emp\l, delegated to, success token} \}$ with eve being the identity of the attacker.

The set $\mathcal{P}$ contains only one unary predicate, $\mathcal{K}$, used for modelling the knowledge of the services and the attacker. Note that the knowledge of each of the services and the attacker are stored separately. In order to avoid unnecessary clumping, we suppress the predicate symbol $\mathcal{K}$.

The attacker $A = (\Omega_{\text{hr}}, \text{I}_A)$ has the initial knowledge $\Omega_{\text{hr}} = \Sigma_C \setminus \{ \text{doc} \cup \{ pk(\cdot) \mid A \in \{ \text{mike, piet, ann, cr, hr, eve} \} \}$, and her intensional knowledge reflects the usual DY message derivation rules. Despite the fact that the attacker’s knowledge set is always finite, she can generate an infinite set of terms by pairing, hashing, signing and encrypting the terms that are known to her (cf. appendix A).

The intensional knowledge for $\text{mike}$, $\text{piet}$, and $\text{ann}$ consists of the $TA$ and $TD$ rules only. The intensional knowledge of the $\text{hr}$ contains the type-1 theory $\{ \text{emp\l}(X) \leftarrow \text{head}(X) \}$, besides $TA$ and $TD$. The intensional knowledge of the $\text{cr}$, in addition to $TA$ and $TD$ contains the type-1 theory $\{ \text{emp\l}(X) \leftarrow \text{head}(X), \text{can\_store}(X) \leftarrow \text{emp\l}(X) \}$, which constitute a simple hierarchical RBAC system, where employees have the right to store documents in the $\text{cr}$, and heads of the office inherit all rights of employees.

Below, we describe the processes executed by the participating services, and their initial knowledge. Recall that capital letters denote variables. To avoid name clashes, variables appearing in different processes should be tagged with process names. The tagging is however omitted in the following to ease the presentation.

Citizen ($\text{mike}$). Mike’s initial knowledge is empty.

$$\emptyset \triangleright s(\{\text{mike, doc}\}_{pk(\text{piet})}, \{h(\text{doc})\}_{\text{mike}})$$

$$r(\{\text{h(mike, doc)\,success\_token\}_{\text{piet}}}) \triangleright \emptyset$$

Mike sends document doc to Piet, in order to be stored in the $\text{cr}$. He then waits to receive a “success” token from Piet. The names $\text{mike}$ and $\text{piet}$, and also doc are constants in Mike’s specification.

Employee ($\text{piet}$). Piet’s initial knowledge is empty.

$$r(\{\text{C, D}\}_{pk(\text{piet})}, \{h(\text{D})\}_{\text{cr}}) \triangleright \emptyset$$

$$\emptyset \triangleright s(\{\text{C, D}\}_{pk(\text{cr})}, \{\text{ann}, h(\text{C, D})\}_{\text{piet}})$$

$$r(\{h(\text{C, D})\}_{\text{cr}}) \triangleright \emptyset$$

$$\emptyset \triangleright s(\{h(\text{C, D}), \text{success\_token}\}_{\text{piet}})$$

After Piet receives a request, from a citizen $\text{C}$, to store a document $D$, he sends a corresponding request to the $\text{cr}$. Then he waits for confirmation (of successful storing) from the $\text{cr}$, after which he notifies the citizen on the completed transaction.

Head ($\text{ann}$). Ann’s initial knowledge is empty.

$$r(\text{E, emp\l status}) \triangleright \emptyset$$

$$\emptyset \triangleright s(\{\text{E, is emp\l, delegated to, hr}\}_{\text{ann}})$$

When Ann receives a request for information on the status of a principal $E$, she replies that the task of providing such information has been delegated to the $\text{hr}$. Ann would typically execute a few instances of this process in parallel.

Human Resources ($\text{hr}$). The initial knowledge of $\text{hr}$ contains all employees and heads of the office. That is, the $\text{hr}$’s initial knowledge is $\{ \text{emp\l(\text{piet}), head(\text{ann})} \}$.

$$r(\text{E, emp\l status}) \triangleright \emptyset$$

$$\emptyset \triangleright s(\{\text{E, emp\l}\}_{\text{hr}})$$

After receiving a request for the status of a principal $E$, the $\text{hr}$ confirms that $E$ is an employee of the office by sending the message $\{\text{E, is emp\l}\}_{\text{hr}}$, only if $\text{emp\l}(E)$ can be derived in the policy engine of the $\text{hr}$.

Central Repository ($\text{cr}$). The $\text{cr}$ service consists of two processes, executed in parallel. The initial knowledge of the $\text{cr}$ is $\{\text{head(ann)}\}$.

Central Repository’s main process.

$$r(\{\text{C, D}\}_{pk(\text{cr})}, \{\text{H, h(C, D)}\}_{\text{cr}}) \triangleright \emptyset$$

$$\{\text{head(\text{H})}\} \triangleright s(\{\text{E, emp\l status}\}_{\text{cr}})$$

$$\{\text{can\_store(\text{E})}\} \triangleright s(\{\text{h(C, D)}\}_{\text{cr}})$$

After receiving the first message from $E$, requesting to store a document $D$ as an employee of the office headed by $H$, the $\text{cr}$ asserts that $H$ is trusted on whether $E$ is an employee of the office, or not. Next, if $H$ is indeed the head of the office, then the $\text{cr}$ asks $H$ for clarifying the employment status of $E$. If the $\text{cr}$, after receiving the third message, ascertains that $H$ has the right to store, then the document is stored (not formalized here), and $E$ is notified.

Central Repository’s delegation handler.

$$r(\{\text{E, is emp\l, delegated to, HR}\}_{\text{hr}}) \triangleright \{\sigma(\text{H, \theta(\text{HR, emp\l(E)})})\}$$

This process receives, independently of the other process,
The formalization given above models the inquiries which the cr conducts via a “broadcast” send. Namely, when the cr sends the message \((E, \text{empl\_status})\), the message is intended for \(H\) (i.e. \(\text{ann}\)), but it is received also by \(F\) (i.e. the \(hr\)). This is because \(\text{ann}\) replies to the message with a delegation certificate, which is consumed by the delegation handler process of the \(cr\). It is thus \(F\) (i.e. the \(hr\)) who actually responds to the message \((E, \text{empl\_status})\) in the \(cr\’s\) main process.

The derivation tree of figure 2 shows how the \(cr\) ascertains Piet’s right to write into the repository; the correspondence between derivation trees and computing closures under Horn clauses is immediate. The rules (e.g. \(TA\) or type-1 theory of the \(cr\)) used in each derivation step are also shown in the figure.

5. DECIDING REACHABILITY

In this section, we give an algorithm for deciding \textsc{reach} in A1 architectures. We start with presenting an encoding from policy statements into (message) terms. Then, we extend the constraint-solving algorithm of Milen and Shmatikov for deciding reachability in the “encoded” A1 architectures.

The purpose of the encoding is to replace the logic programs of the participating services in an A1 architecture with the derivation rules of the DY model. Then, intuitively, the attacker and all the services would be equipped with the reasoning power of the DY model, which is well understood and comes with decision algorithms for reachability.

5.1 Encoding policy level computations

Below, to simplify the presentation, we suppress the predicate symbol \(K\) from facts, and work directly with infons; indeed \(K\) is the only predicate symbol in A1 architectures. The encoding consists of two functions: \(\zeta\) which maps infons to \(T_{\text{env}(\theta)}\), and \(\mathcal{E}\) which maps infons to guards. We start with an initial encoding for trust application only. Then, we extend the encoding to trust delegation and type-1 theories.

\(TA\) only. We recursively define the encoding for infon \(i\):

\[
\zeta(i) = \begin{cases} 
\theta(\bar{A}, \zeta(X)) & \text{if } i = \sigma(\bar{A}, X) \\
\theta(\bar{A}, \zeta(X)) & \text{if } i = \theta(\bar{A}, X) \\
& \text{otherwise}
\end{cases}
\]

where \(\zeta: \text{Agents} \to \bar{\text{Agents}}\) is a bijection which associates a unique name to each element of \(\text{Agents}\). Elements of \(\bar{\text{Agents}}\) belong to \(T_{\text{env}(\theta)}\), and are defined solely for the encoding function \(\zeta\), i.e. they do not appear in the specifications of architectures. In particular, the attacker does not have the private keys associated to the members of \(\text{Agents}\).

Here, the encoding of the infon \(\sigma(a,x)\) is the ciphertext \(\{\bar{x}, \text{sig}(a,x)\}_{\theta(a,x)}\) from which the infon \(x\) (i.e. what service a said) can be obtained using the symmetric decryption (Sdec) rule of DY only if the key \(\theta(a,x)\) (i.e. \(a\) is trusted on \(x\)) is obtained first. This indicates that if \(TA\) is applicable on a set of facts at the policy level, then the DY rule

\[
K(X) \leftarrow K(\{X\}_K), K(K) \text{ Sdec}
\]

is applicable to the terms resulting from the encoding. Intuitively, the role of \(\text{sig}\) is to ensure that terms of the form \(\{x, \text{sig}(a,x)\}_{\theta(a,x)}\) can be constructed using the DY rules only if a corresponding \(\sigma(a,x)\) can be derived in the policy level. Recall that the attacker does not know the private key of \(a\).

\(TA, TD\) and type-1 theories. In order to include \(TD\) and type-1 theories in the encoding, we define an expansion function \(\mathcal{E}\) that for any atom returns a guard. We motivate the expansion function via a simple example. Suppose the fact \(\theta(a,i)\) is present in the policy engine of a service, and the guard \(\theta(a,\theta(b,i))\) is to be evaluated. The \(TD\) rule implies that the guard can be derived, while there is no corresponding inference tree (in the DY model) for \(\zeta(\theta(a,\theta(b,i)))\), given \(\zeta(\theta(a,i))\). The set of infons which yield \(\theta(a,\theta(b,i))\) via applying only the \(TD\) rule is however finite. This finite set of infons can be seen as a guard, namely \(\{\theta(a,\theta(b,i))\} \lor \{\theta(a,i)\}\). The fact that \(\theta(a, i)\) yields \(\theta(a, \theta(b,i))\) in the policy engine is reflected in the DY model by: either \(\zeta(\theta(a, \theta(b,i)))\) or \(\zeta(\theta(a,i))\), or both, are obtained from \(\zeta(\theta(a,i))\).

The expansion function \(\mathcal{E}(P,i)\) is defined for finite sets of Horn clauses \(P\) and \(Q\), and infon \(i\):

\[
\mathcal{E}_P(\theta, i) = \{i\} \\
\mathcal{E}_P(\{r \leftarrow r_1, \ldots, r_l \lor Q', i\}) = \\
\{\mathcal{E}_P(Q', i)\} \cup \{\mathcal{E}_P(P, r_1 \rho) \cup \cdots \cup \mathcal{E}_P(P, r_l \rho)\} \quad \text{if } i = r \rho \\
\mathcal{E}_P(Q', i) \quad \text{if } \neg \exists \rho. i = r \rho
\]

where \(\cup\) distributes over \(\lor\), i.e. \(S \cup (S_1 \lor S_2) = (S_1 \lor S_2) \cup S = (S_1 \cup S) \lor (S_2 \cup S)\). Here \(Q\) is a support theory used only to ensure that the expansion of any infon results in a finite set; see lemma 1 below.

**Lemma 1.** Let \(P = P_1 \cup \{TD\}\), with \(P_1\) being the type-1 theory in a service of an A1 architecture. Then, \(\mathcal{E}_P(P,i)\) is a finite set for any infon \(i\).
Proof. Immediate, since the dependency graph of $P^1$ is acyclic. $P^1$ does not contain $\theta$, and the Horn clause which encodes $TD$ strictly decreases the number of $\theta$ functions. □

We write $E(i)$ for $E(P, i)$, when $P$ is clear from the context. We write $g \subseteq E(i)$ if $E(i) = g_1 \lor \cdots \lor g_n$ with $n \geq 1$.

EXAMPLE 2. Consider the infon $i = \text{can\textunderscore read}(a, file)$ along with the type-1 theory of example 1. Then,

$$E(i) = \{\text{user}(a), \text{public\textunderscore file}\} \lor \{\text{admin}(a), \text{public\textunderscore file}\} \lor \{\text{admin}(a), \text{classified\textunderscore file}\} \lor \{\text{can\textunderscore read}(a, file)\}.$$

Write $E(i) = g_1 \lor g_2 \lor g_3$ with $g_1 = \{\text{user}(a), \text{public\textunderscore file}\}$, etc. The guard $E(i)$ is interpreted as: $\text{can\textunderscore read}(a, file)$ holds, i.e. $a$ can read $file$ in example 1, if at least one of the following conditions holds: $g_1$ $\text{user}(a)$ and $\text{public\textunderscore file}$ are known, or $g_2$ $\text{admin}(a)$ and $\text{public\textunderscore file}$ are known, or $g_3$ $\text{admin}(a)$ and $\text{classified\textunderscore file}$ are known, or $g_4$ $\text{can\textunderscore read}(a, file)$ is known via an inference outside the RBAC system of example 1. Similarly, we get $E(\text{can\textunderscore write}(a, file)) = \{\text{admin}(a)\} \lor \{\text{can\textunderscore write}(a, file)\}$.

We refine the function $\zeta$ (introduced above) by incorporating the expansion function $E$ into $\zeta$. This intuitively ensures that $\zeta(i)$, for infon $i$, is obtainable from $\zeta(\sigma(a, i))$ if there exist at least one $g \subseteq E(\theta(a, i))$ such that $\zeta(g)$ can be obtained first. Hence, we define:

$$\zeta(i) = \begin{cases} \{\zeta(X), \text{sig}(\bar{A}, \zeta(X))\} & \text{if } i = \sigma(A, X) \\ \text{sig}(\bar{A}, \zeta(X)) & \text{if } i = \theta(A, X) \\ \emptyset & \text{otherwise} \end{cases}$$

Here, $A \|_{k_1 \lor \cdots \lor k_2} \ldots \|_{k_t}$ stands for the tuple $\{x \|_{k_1}, \ldots, x \|_{k_t}\}$, function $\zeta$ distributes over $\lor$, and $P = P^1 \cup \{TD\}$ with $P^1$ being the type-1 theory at hand. For a finite set of infons $g$, $\zeta(g)$ is defined as the concatenation of $\zeta(i)$, for all $i \in g$. We remark that elements of $E(P, \theta(A, X))$ in the definition of $\zeta$ are singletons. This is because in any $A_1$ architecture, $E(P, \theta(A, X)) = E(TD) \cup \{\text{can\textunderscore read}(a, file)\}$, as $\theta$ does not appear in $P^1$.

**Correctness of the encoding.** We remark that the purpose of the proposed encoding is to replace the logic programs of services with the derivation rules of the Dolev-Yao model.

The following theorem ensures that if a policy fact is derivable in the logic program of a service, then its corresponding encoded term can be derived using the DY inference rules, and vice versa. We consider the standard DY capabilities for the term algebra $T_{\text{DY}(\Gamma)}$, which comprises both infon and message constructors. That is, the infon constructors are seen as uninterpreted functions, while message constructors (e.g. $\|_{\theta}$) have their standard meaning in the DY model.

In the rest of the paper, $T \vdash u$ denotes that $u$ is derivable from a set of terms $T$ with the standard DY inference rules as formalized, e.g., in appendix A and [15].

**Theorem 1.** Let $P$ be the intensional knowledge of a service in an $A_1$ architecture, with $P = \{TA\} \cup Q$, $Q = \{TD\} \cup P^1$ and $P^1$ being a type-1 theory. For any (ground) infon $f$ and finite set of (ground) infons $G$,

$$K(f) \in [K(G)]^P \iff \exists g \subseteq E_Q(Q, f) \cdot \zeta(G) \vdash \zeta(g),$$

where $K(G)$ stands for the set $\{K(f) \mid f \in G\}$.

Proof. Fix the policy set $P$. We write $G \models f$ for $K(f) \in [K(G)]^P$, suppress $K$ when confusion is unlikely, and write $E(f)$ for $E_Q(Q, f)$. Below, we talk about proof trees for $f$, given $G$. The correspondence between finding proof trees and computing closures is immediate. The proof is split into two directions.

$$\Rightarrow$$ We use structural induction on proof trees for $f$, given $G$.

If $f \in G$, then the implication is trivial. Otherwise, consider the last rule applied in the proof tree:

- (TA) Then $G \models \sigma(a, f), \theta(a, f)$, for some $a \in \text{Agents}$. By induction hypotheses,

  $$\exists s \subseteq E(\sigma(a, f)), t \subseteq E(\theta(a, f)), \zeta(G) \vdash \zeta(s), \zeta(t).$$

  Observe that $s = \sigma(a, f)$. The term $\zeta(\sigma(a, f))$ is the tuple $\{\zeta(f), \text{sig}(\bar{\pi}, \zeta(f))\} \in E(\sigma(a, f))$. Since $t \subseteq E(\theta(a, f))$, through unpairing, we obtain the ciphertext $\{\zeta(f), \text{sig}(\bar{\pi}, \zeta(f))\} \cdotp \zeta(i)$ from $\zeta(\sigma(a, f))$. From $\zeta(G) \vdash \zeta(t)$, by applying the $\text{Sdec}$ rule and unpairing we get $\zeta(G) \vdash \zeta(f)$. Clearly $f \subseteq E(f)$.

- (TD) Then $f = \theta(a, (b, i))$ for some $a, b \in \text{Agents}$ and $i \in \text{Infons}$, and $G \models \theta(a, i)$. By induction hypotheses, $\exists \bar{v} \subseteq E(\theta(a, i)), \zeta(G) \vdash \zeta(t)$. Now, the claim follows since for any infon $t, t \subseteq E(\theta(a, i))$ implies $t \subseteq E(\theta(a, (b, i)))$.

- (Type-1) Let $R = r_1 \cdots r_t \in P^1$ be the last rule applied. Then $f = \bar{r} \rho$ and $G \models r_1 \rho, \cdots, r_t \rho$, for some grounding substitution $\rho$ (cf. condition (b) in definition 2). By induction hypotheses, $\exists \bar{r}_1 \subseteq E(r_1 \rho), \cdots, \bar{r}_t \subseteq E(r_t \rho), \zeta(G) \vdash \zeta(\bar{r}_1), \cdots, \zeta(\bar{r}_t)$. By definition of $E$, $\{\bar{r}_1, \cdots, \bar{r}_t\} \subseteq E(f)$, hence follows the claim.

$$\Leftarrow$$ First, we claim that $\zeta(G) \vdash \zeta(g)$ implies $G \models g$. Notice that the $\zeta(g)$ is either of the form $\{\zeta(x), \text{sig}(\bar{\pi}, \zeta(x))\} \in E(\theta(a, x))$, or of the form $\{\zeta(x)\} \in E(\theta(x), a)$, with $x$ being an infon constructor. The claim follows by case analysis on the DY attacker’s message (de)composition abilities. In particular, note that (1) to fabricate $\{\zeta(x), \text{sig}(\bar{\pi}, \zeta(x))\} \in E(\theta(a, x))$, the attacker needs to construct $\text{sig}(\bar{\pi}, \zeta(x))$, which is impossible as the attacker does not own the private key for any $a \in \text{Agents}$, and (2) infon constructors are uninterpreted functions in the DY model, i.e. they can neither be applied by the attacker, nor their application can be deconstructed. The other cases are straightforward; we thus omit them here. Finally, notice that if $G \models g$ and $g \subseteq E(f)$, then $G \models f$; hence follows the claim.

This completes our proof. □

5.2 A constraint-solving algorithm for deciding reachability
We begin with a brief description of Millen and Shmatikov's constraint solving algorithm for deciding reachability in cryptographi
c protocols [15]. Recall that participants are spec-
ied as sequences of communication (i.e. send and receive)
ev ents in [15], and the reachability problem, given a mes-
s sage \( s \), asks whether there exists a reachable configuration \( z \) of the protocol where \( T(z) \vdash s \), with \( T(z) \) being the at-
tacker’s knowledge in \( z \).

The algorithm of [15] searches the finite set of interleavings of (symbolic) actions performed by the participants and a fictitious test process, which receives \( s \) and then sends stop to the search algorithm. For each interleaving, a sequence \( C \) of attacker constraints is constructed. An attacker con-

straint is a pair \( \langle m : T \rangle \), where \( m \) is a term that the attacker should derive from the set of terms \( T \), using her inference capabilities. The constraint sequence is built for each interleaving as: when a participant sends a message term, the term is added to the attacker term set, and when a receive action occurs, a constraint \( \langle m : T \rangle \) is enqueued to \( C \), with \( m \) being the term that is to be received and \( T \) is the current attacker term set.

A solution for constraint sequence \( C = \langle m_1 : T_1 \rangle \cdots \langle m_s : T_s \rangle \), with \( m_s = s \), is a (grounding) substitution \( \sigma : \text{var}(\langle m_1 : T_1 \rangle \cdots \langle m_s : T_s \rangle) \rightarrow T_\Sigma(\emptyset) \) such that \( T_\sigma \vdash m_1 \sigma \), for \( 1 \leq j \leq i \); here, \( \text{var}(c_1 \cdots c_s) \) is the set of variables appearing in the constraints \( c_1 \cdots c_s \). In our presentation, therefore, we account for partial executions as well, cf. [6]. Millen and Shmatikov’s algorithm applies a number of reduction rules which reduce \( C \) to a sequence of immediately (un)satisfiable constraints. In the following, we refer to their reduction procedure as MSReduce. If MSReduce does not succeed for \( C \) (i.e. \( C \) is unsatisfiable), next interleaving is considered, until all the interleavings are exhausted. If one of the constraint sequences is satisfiable, then the (suppos-
edly) secret message \( s \) is revealed to the attacker. That is, an attack is found. Otherwise the protocol is correct, i.e. \( s \) is not revealed to the attacker, for the instantiation at hand.

The following two observations enable us to use Millen and Shmatikov’s procedure (with minor extensions) for deciding reachability for \( A_1 \) architectures:

1. Checking guards in, and making updates to, policy engines of the services can be emulated by communication actions. Namely, sending an infor to the knowledge set of a service reflects updating the policy engine of that service, while receiving an infor derived from the knowledge set reflects querying the policy engine of the service for evaluating a guard.

2. The encoding presented in section 5.1 entails that the same inference rules which are used for the attacker (namely the standard DY message derivation capabilities) can model the computations of the participants at their policy level. Therefore, the send and receive actions which would emulate checking guards and up-
dating knowledge sets of services (mentioned above) can be treated with the constraint solving procedure that one would use for the attacker knowledge (here, MSReduce).

Algorithm 1 Constraint solving for deciding reachability in \( A_1 \) fragment

REQUIRES: \( \text{Reach}(\text{arch}, a, f) \), \( \text{arch} = (\Sigma, V, P), II, A \)

expand(II)

\( I := \text{interleavings}(II, a, f) \) for all \( i \in I \) do

\( \text{flatten}(i) \)

\( C := \text{constraint}_\text{sequence}(i) \)

\( \text{trace} := \text{MSReduce}^\text{a}(C) \)

if \( \text{trace} \neq \emptyset \) then

return (reach : \text{trace})

return (unreach)

Algorithm 1 details our constraint solving procedure for de-
ciding reachability in \( A_1 \) architectures. The input to the algorithm is a reachability problem \( \text{Reach}(\text{arch}, a, f) \), with \( \text{arch} = (\Sigma, V, P), II, A \) being an \( A_1 \) architecture. The algo-

rithm either returns a trace witnessing \( \text{Reach}(\text{arch}, a, f) = T \), or returns unreach if \( \text{Reach}(\text{arch}, a, f) = T \).

In algorithm \( 1, \text{expand}(II) \) expands the guards in each pro-
cess, for all the services in \( II \). A guard \( g_1 \cdots g_n \), in a service with policy \( P \), is expanded to \( \bigvee_{i < j < n} E_Q(Q, g_i) \), where \( Q = P \setminus \{ TA \} \) and \( E_Q(Q, g_i) \) for a finite set of inforns \( g \) is defined as the union of \( E_Q(Q, i) \), for all \( i \in g \). Recall that \( \cup \) distributes over \( \lor \) (cf. section 5.1). The expand procedure thus rewrites guards into guards. For example, the guard \( \{ a, b \} \) in a service with intensional knowledge \( \{ TA, TA, a \leftarrow c, b \leftarrow d \} \) is expanded to \( \{ a, b \} \lor \{ c, b \} \lor \{ a, d \} \lor \{ c, d \} \).

The procedure \( \text{interleavings}(II, a, f) \) computes the finite set of interleavings of events of the participants in \( II \). Furthermore, this procedure \((1)\) adds a testing process whose sole purpose is to indicate that the search has reached a configu-
r ation in which \( f \in \Omega(a)^T \) has been reached \( \text{Reach}(\text{arch}, a, f) \); that is, if \( a = A \) and \( f = K(m) \), then the testing process simply receives \( m \) and then sends stop to the search algorithm. If \( a \neq A \), then the testing process \( \{ f \} \rightarrow \text{stop} \) is added to service \( a \).

(2) The \( \text{interleavings} \) procedure treats the dis-

junction operator \( \lor \) inside guards as branching points, e.g. the

interleaving \( e_1 \cdot (g \lor g) \rightarrow s(m) \cdot e_2 \), with \( e_1, e_2 \in E^* \), gives rise to two interleavings \( e_1 \cdot (g \rightarrow s(m)) \cdot e_2 \) and \( e_1 \cdot (g') \rightarrow s(m) \cdot e_2 \). Consequently, no \( \lor \) appears in guards for any interleaving \( \iota \in I \) in algorithm 1.

For each interleaving, the procedure \( \text{flatten} \) intuitively “flattens” the two levels of specification into one. That is, a sequence of events (i.e. guarded sends, and receives coupled with updates) is translated into a sequence of annotated send and receive actions. The annotations indicate the knowledge set with which the communication is carried out: \( \text{fork} \) for network communications through the attacker, and \( \pi \in II \) for each service \( \pi \).

\( \{ e_1, \ldots, e_t \} \rightarrow s(m) \) maps to \( r^\ast(\xi(a_1), \ldots, \xi(a_t)) \cdot s^\ast(m) \)

\( (r(m) \triangleright \{ a_1, \ldots, a_r \}) \) maps to \( r^\ast(m) \cdot s^\ast(\xi(a_1), \ldots, \xi(a_r)) \)

The annotated send and receive actions are used in the redu-
cution procedure MSReduce, which is a barely syntactic modi-
fication of MSReduce. The modification consists in allo-
cating one term set for each service in \( II \), and one set for the attacker. This is in contrast to MSReduce where only one
term set, denoting the attacker knowledge, is considered. In \textsc{MSReduce} \textsuperscript{6d}, annotated sends and receives communicate with the term set that is determined by their annotation.

After flattening, the procedure \textit{constraint\_sequence} generates a constraint sequence for the interleaving (as it is done in [15]). The resulting constraint sequence is fed to the procedure \textsc{MSReduce} \textsuperscript{6d}.

\textbf{Example 3.} Consider a file server service \(fs\), with intensional knowledge \(P = \{TA, TD\} \cup P^r\), where \(P^r\) is the type-1 theory of example 1. A typical event of the main process of the \(fs\) service is \(\{\text{can\_read}(A, F)\} \triangleright s(F)_{\text{pk}(A)}\), where \(A\) and \(F\) denote, respectively, a client of the service and a file stored on \(fs\). For this event the \textit{expand} procedure returns \(\{\text{user}(A), \text{public}(F)\} \cup \{\text{admin}(A), \text{public}(F)\} \cup \{\text{admin}(A), \text{classified}(F)\} \cup \{\text{can\_read}(A, F)\} \triangleright s(F)_{\text{pk}(A)}\): see example 2.

The interleavings procedure then creates a branch for each \(g \in A\) the resulting guard, while interleaving this event with other events of the architecture. For each of the branches the \textit{flatten} procedure maps the events into communication actions. For example, the guarded send \(\{\text{user}(A), \text{public}(F)\} \triangleright s(F)_{\text{pk}(A)}\) maps to \(s'(\text{user}(A), \text{public}(F)) \triangleright s\left(F_{\text{pk}(A)}\right)\).

The following theorem states that algorithm 1 is terminating on decision problems for \(A_1\) architectures; moreover the algorithm is correct (i.e. sound and complete) w.r.t. the semantics of \(A_1\) architectures.

\textbf{Theorem 2.} Given decision problem \textsc{Reach}(\textit{arch}, \textit{a}, \textit{f}), with \textit{arch} being an \(A_1\) architecture, algorithm 1 terminates, and returns a trace iff \textsc{Reach}(\textit{arch}, \textit{a}, \textit{f}) = T.

The proof of theorem 2 is relegated to appendix B. Below, we sketch the main idea of the proof. Let \(KS = (S, S^0, T)\) be the Kripke structure induced by \textit{arch}. A trace in \(KS\) is a sequence \(z_0 e_1 z_1 \cdots e_n z_n\), where \(z_0 = S^0\) and the system evolves from \(z_{i-1}\) into \(z_i\) when event \(e_i\) occurs, as defined in section 3.2. A symbolic trace \(st\) is a trace which may contain non-ground terms. We say \(st\) is realizable in \(KS\) iff there exists a grounding substitution \(\sigma\), such that \(st\sigma\) is a trace in \(KS\). Given a symbolic trace \(st\) and a sequence of constraints \(C\), we define their correspondence inductively: if \(st = st_0\), then the empty sequence, i.e. \(C = \sigma = 0\), corresponds to \(st\). Now, let \(st = z_0 e_1 \cdots z_n e_n\), with \(n \geq 0\), and \(C'\) be a sequence of constraints that corresponds to \(z_0 \cdots z_n\). If \(e = g \triangleright s(m)\), then any \(C = C' \cup \{\zeta(g) : \zeta(\omega_a^{e_n})\}\) corresponds to \(st\), where \(\pi\) is the service that performs \(e\), \(g \in E(g)\) is calculated w.r.t. the intensional knowledge of \(\pi\), and \(\omega_a^{e_n}\) refers to the knowledge of \(\pi\) at symbolic configuration \(z_n\). The correspondence between event \(e\) and constraint \(\zeta(g) : \zeta(\omega_a^{e_n})\) hinges upon theorem 1, which tells us that the encoding function \(\zeta\) is such that \(\zeta(G) \cup \zeta(g)\) for at least one \(g \in E(g)\) if \(K(G) \in |K(G)|^p\), given any \(G \subseteq A_{3D}\). If \(e = r(m) \triangleright u\), then \(C = C' \cup \{m : \omega_a^{e_n}\}\) corresponds to \(st\), where \(\omega_a^{e_n}\) refers to the attacker knowledge at symbolic configuration \(z_n\). We remark that for the attacker knowledge the (suppressed) predicate \(K\) ranges directly over messages.

From the definition above, clearly there are finitely many constraint sequences (created due to the \(\lor\) operator in guards) corresponding to any symbolic trace \(st\). The proof of theorem 2 intuitively goes by showing that for each symbolic trace \(st\), created by an interleaving of events of services in \textit{arch}, \(st\) is realizable in \(KS\) iff a constraint sequence corresponding to \(st\) is satisfiable in algorithm 1.

We proceed with the CRP case study.

\subsection{Verification results for CRP}

We have developed a tool which implements algorithm 1. The tool is written in Prolog, and extends the constraint solver developed by Millen and Shmatikov [15]. We have used the tool to verify a number of properties on the CRP case study (specified in section 4.1), namely: (1) \textit{Executability}. We replaced the last event in the specification of the citizen with

\[ r([\text{ mike, doc}],[\text{success\_token}]) \triangleright \{\text{stored(doc)}\} \]

and checked if \textsc{Reach}(\textit{crp, mike, stored(doc)}) = T; that is, whether \textit{mike} knows that his document has been successfully stored in the \textit{cr}. The tool returns a trace showing that this property indeed holds in the CRP architecture. (2) \textit{Secrecy}. We have checked if \textsc{Reach}(\textit{crp, eve, doc}) = T, i.e. whether the attacker \textit{eve} can discover \textit{doc}. No (attack) trace is found. (3) \textit{Safety}. We have checked whether the attacker \textit{eve} can obtain the right to store in the \textit{cr}. That is, if \textsc{Reach}(\textit{cr, can\_store(eve)}) = T. No (attack) trace is found.

It is worth mentioning that in \(A_1\) architectures, e.g. the formalization of CRP in section 4.1, only a finite number of services are allowed. Our decidability result would in fact fall apart if an unbounded number of services were considered in \(A_1\) architectures. This immediately follows from the undecidability of reachability in security protocols with an unbounded number of sessions.

\section{Conclusion}

We have presented a language for formal specification of service-oriented architectures. The language allows us to specify communication level events, policy level decisions, and the interface between the two. We have shown that the reachability problem is decidable for a fragment of architectures specified in the language. The decidable fragment is of immediate practical relevance.

In this decidable fragment, the policies of the (honest) services are limited to trust application and trust delegation rules, besides a finite set of Horn clauses of type-1. Type-1 Horn theories are characterized here by placing certain syntactic conditions on the form of the clauses. As we show in the paper, these conditions are indeed sufficient; they are however in general not necessary for deciding reachability. We intend to investigate how, without undermining our decidability result, the policies of honest services can be extended to theories beyond type-1 which are interesting in practice. This is left for future work.

Revocation of rights is not expressible in a natural way in authorization logics which are based on Horn theories, cf. [14]
for an overview. We are currently working towards accommodating rights revocation in our language. Investigating decidability issues in the presence of revocation is left for future work.

We (similar to, e.g., dkal) see “know” as a predicate, as opposed to a modality as it is common in epistemic logics. This causes a discrepancy between the syntactic form of “knowledge”, represented by predicates, and the semantic notion of knowledge. It must therefore be interesting to investigate how our proposed formalism (or, dkal) can be connected to epistemic models, such as knowledge-based programs [10] and dynamic epistemic logic [8].

Acknowledgements
We are grateful to David Basin, Samuel Burri, Silvio Ranise and Eugen Zalinescu for their comments on this paper. The work has been supported by AVANTSSAR, FP7-ICT-2007-1 Project no. 216471.

7. REFERENCES

APPENDIX
A. ATTACKER MODEL
Below, we list the capabilities of the Dolev-Yao attacker model. The following rules describe the intentional knowledge (i.e. the deduction capabilities) of the attacker, cf. [5].

\begin{align*}
\text{Analysis} & \\
{x} & \leftarrow \langle x, y \rangle & \phi_{proj1} \\
y & \leftarrow \langle x, y \rangle & \phi_{proj2} \\
x & \leftarrow \{x\}_{pk(eve)} & \phi_{pk} \\
x & \leftarrow \{x\}_{pk(eve)} & \phi_{dec} \\
{x} & \leftarrow \{x\}_{pk(eve)}, y & \phi_{edc} \\
\text{Synthesis} & \\
(x, y) & \leftarrow \langle x, y \rangle & \phi_{pair} \\
\{x\}_{y} & \leftarrow \langle x, y \rangle & \phi_{enc} \\
\{y\}_{x} & \leftarrow \langle x, y \rangle & \phi_{dec} \\
b(x) & \leftarrow \langle x \rangle & \phi_{hash} \\
\sigma(pk(eve), x) & \leftarrow \langle x \rangle & \phi_{sig} \\
\end{align*}

B. PROOF OF THEOREM 2
Below, we present the proof of theorem 2. The proof relies upon a lemma on correctness and termination of MSReduce\textsuperscript{eq}.

\text{Proof of Theorem 2.} Termination of algorithm 1 is immediate, as the procedure interleavings produces a finite number of (symbolic) interleavings, and all the functions applied to interleavings, in particular MSReduce\textsuperscript{eq} (see lemma 2, below) and expand (due to lemma 1), terminate in finitely many steps.

Notice that algorithm 1 exhaustively considers all possible symbolic traces of arch, and for each symbolic trace stå, the algorithm considers all constraint sequences corresponding to stå. We write stå for the set of all constraint sequences which corresponds to stå.

The proof is split into two directions.

⇒ Assume \text{REACH}(arch, a, f) = T. Then, there exists a symbolic trace stå = z_{e1} \cdots z_{en} z_{n}, and a nonempty set of substitutions \mathcal{S} = \{ \sigma_1, \sigma_2, \ldots \} where stå\sigma_j is a trace in KS, for any \sigma_j \in \mathcal{S}. Then, there exists at least one constraint sequence C \in stå, which is satisfiable by any substitution \sigma_j \in \mathcal{S}, and that is considered by algorithm 1. It now remains to show that MSReduce\textsuperscript{eq} returns a witness trace for C. This holds due to lemma 2, below.
Let $C$ be a constraint sequence generated in algorithm 1. Then, $C \in st$, for some symbolic trace $st$. Suppose MSReduce$_{u}$ returns a witness trace, showing that $C$ is satisfiable under (non-ground) substitution $\rho$ (see lemma 2 on applicability of MSReduce$_{u}$, below). By theorem 1 and the fact that solving constraint $(m; T)$ implies $T \vdash m$, it follows that $st$ is a trace in KS, for any ground substitution $\sigma$ that refines $\rho$. That is, Reach$(arch, a, f) = T$.

This completes the proof.

Lemma 2, below, concerns the applicability of MSReduce$_{a}$. Given a reduction system for constraint sequences, we write $C \rightarrow C'$ iff $C$ is reduced, using a rule in the reduction system, to $C'$. A reduction system is

- **terminating** iff it admits no infinite reductions;
- **sound** iff $C \rightarrow C'$ implies that for any $\sigma$ that is a solution of $C'$, $\sigma$ is also a solution of $C$;
- **complete** iff for any $\sigma$ that is a solution of $C$, there exists a reduction step $C \rightarrow C'$ such that $\sigma$ is a solution of $C'$.

**Lemma 2.** MSReduce$_{a}$ is terminating, sound and complete for the constraint sequences generated in algorithm 1.

We prove lemma 2 by revising the proofs of termination and correctness (i.e. soundness and completeness) of MSReduce, originally presented in [15]. We start by giving an overview of MSReduce in appendix B.1. There, we also single out the differences between MSReduce and MSReduce$_{a}$. Then, the proofs of termination and correctness of MSReduce$_{a}$ are given, respectively, in appendices B.2 and B.3.

**B.1 Overview of MSReduce**

**Extended attacker model.** The attacker model considered in [15] is based on the standard Dolev-Yao attacker model, as formalized in appendix A. In addition, Milner and Shmatikov consider an encryption hiding operator, here denoted $\| \cdot \|_y$, that serves as a technical means to avoid non-termination of the constraint-solving algorithm. Consequently, the following attacker capabilities are also considered:

\[
\begin{align*}
|c|_y & \leftarrow \|x\|_y, & \Phi_{\text{open}} \\
\|x\|_y & \leftarrow \|c\|_y, & \Phi_{\text{hide}}
\end{align*}
\]

Given a set of messages $T$ we denote by $F(T)$ the set of messages that the attacker can derive from $T$ using her capabilities. In terms of the formalization introduced in the paper, $F(T) = \{T\}^{T_{\text{ks}}}$.

**Reduction procedure.** In the following we give a short description of the MSReduce algorithm (see algorithm 2). The algorithm takes an initial constraint sequence $IC$ as input, and builds the tree of all possible reductions for $IC$. Each node is labelled by a pair $(C, \sigma)$ whose first element is a constraint sequence, and the second element is a substitution for the variables in $C$. Starting from the root $(IC, \emptyset)$, the tree is explored with a depth first search. Notice that the algorithm 2 uses a stack structure, rather than a tree structure, as it is usual in the depth first search. The exploration terminates successfully (i.e. node $(C, \sigma)$ corresponds to an attack) when $C$ is simple; that is, every constraint in $C$ is of the form $(V; T)$ where $V$ is a variable. Simple constraint sequences (under the monotonicity and origination assumptions, see below) are immediately solvable.

**Algorithm 2 MSReduce**

**Requires:** initial constraint sequence $IC$

\[
\begin{align*}
\text{stack} & := \emptyset \\
\text{stack.push}(IC, \emptyset)
\end{align*}
\]

**repeat**

\[
\begin{align*}
& (C, \sigma) := \text{stack.pop} \\
& \text{let } c = (m; t) \text{ be the first constraint in } C \\
& \text{s.t. } m \text{ is not a variable} \\
& \text{if } c \text{ not found then} \\
& \text{return } \text{(satisfiable : } (C, \sigma)) \\
& \text{apply rule (elim) to } c \text{ until no longer applicable} \\
& \text{for all } r \in R \text{ do} \\
& \text{if } r \text{ is applicable to } C \text{ then} \\
& \text{stack.push}(r(C, \sigma)) \\
& \text{until stack.empty} \\
& \text{return } \text{(unsatisfiable)}
\end{align*}
\]

Initially, the root of the tree is set to the pair $(IC, \emptyset)$ (i.e. $(IC, \emptyset)$ is the only element of the stack). Then, the tree is explored as follows. A node $(C, \sigma)$ is popped from the stack, and the active constraint (that is, the first constraint $c = (m; t) \in C$ with a non-variable $m$) is picked. If no such constraint exists in $C$, then $C$ is simple hence the node is returned as a solution for the constraint sequence. If $c$ is found, then all occurrences of stand-alone variables are removed from the term set $T$. Subsequently, for any applicable reduction rule $r$ (see paragraph Reduction rules, below), the new node resulting from the application of $r$ to the current node is pushed on top of the stack. When no nodes are left in the stack, all possible reductions have been considered, so the algorithm returns “unsatisfiable” and terminates.

**Reduction rules.** In the following we present the reduction rules used by MSReduce. The rules read from top to bottom, that is, a rule $r$

\[
\frac{(C_{<} \cdot c \cdot C_{>}, \sigma)}{(C_{<} \cdot c' \cdot C_{>}', \sigma')} \quad r
\]

says that constraint sequence $C_{<} \cdot c \cdot C_{>}$, where $c$ is the active constraint and $C_{<}$ and $C_{>}$ are respectively the constraints preceding and following $c$, is reduced to $C_{<}' \cdot c' \cdot C_{>}'$, and substitution $\sigma$ is refined by $\sigma'$.

\[
\begin{align*}
(C_{<} \cdot (m; T \cup V) \cdot C_{>}, \sigma) & \quad \text{elim} \\
(C_{<}' \cdot (m; T) \cdot C_{>}', \sigma') & \\
\text{where } v \text{ is a variable}
\end{align*}
\]
\[
\frac{(C_\prec \cdot (m:T) \cdot C_\succ, \sigma)_{\text{un}}}{(C_\prec \cdot (m:T) \cdot C_\succ, \sigma)}
\]

where \( \tau = \text{mgu}(m, t), t \in T \)

\[
\frac{(C_\prec \cdot ((m_1, m_2):T) \cdot C_\succ, \sigma)}{(C_\prec \cdot (m_1:T) \cdot (m_2:T) \cdot C_\succ, \sigma)}_{\text{pair}}
\]

\[
\frac{(C_\prec \cdot \langle h(m):T \rangle \cdot C_\succ, \sigma)}{(C_\prec \cdot \langle m:T \rangle \cdot C_\succ, \sigma)}_{\text{hash}}
\]

\[
\frac{(C_\prec \cdot \langle \{m\}_k:T \rangle \cdot C_\succ, \sigma)}{(C_\prec \cdot \langle k:T \rangle \cdot \langle m:T \rangle \cdot C_\succ, \sigma)}_{\text{penc}}
\]

\[
\frac{(C_\prec \cdot \langle \{m\}_k:T \rangle \cdot C_\succ, \sigma)}{(C_\prec \cdot \langle k:T \rangle \cdot \langle m:T \rangle \cdot C_\succ, \sigma)}_{\text{senc}}
\]

\[
\frac{(C_\prec \cdot \langle \text{sig}(pk(\text{eve})):T \rangle \cdot C_\succ, \sigma)}{(C_\prec \cdot \langle m:T \rangle \cdot C_\succ, \sigma)}_{\text{sig}}
\]

\[
\frac{(C_\prec \cdot \langle m:T \cup \{t_1, t_2\} \rangle \cdot C_\succ, \sigma)}{(C_\prec \cdot \langle m:T \cup \{t_1, t_2\} \rangle \cdot C_\succ, \sigma)}_{\text{split}}
\]

\[
\frac{(C_\prec \cdot \langle m:T \cup \{t\} \rangle \cdot C_\succ, \sigma)}{(C_\prec \cdot \langle m:T \cup \{t\} \rangle \cdot C_\succ, \sigma)}_{\text{pdec}}
\]

\[
\frac{(C_\prec \cdot \langle m:T \cup \{t\} \rangle \cdot C_\succ, \sigma)}{(C_\prec \cdot \langle m:T \cup \{t\} \rangle \cdot C_\succ, \sigma)}_{\text{ksub}}
\]

where \( \tau = \text{mgu}(k, pk(\text{eve})), k \neq pk(\text{eve}) \)

\[
\frac{(C_\prec \cdot \langle m:T \cup \{\{k\}_k\} \rangle \cdot C_\succ, \sigma)}{(C_\prec \cdot \langle k:T \rangle \cdot \langle m:T \cup \{k, t\} \rangle \cdot C_\succ, \sigma)}_{\text{pdec}}
\]

**Properties of MSReduce.** For applying MSReduce to a sequence of constraints \( C \), it is required that [15]:

- (Monotonicity) If constraint \( \langle m:T \rangle \) precedes \( \langle m':T' \rangle \) in \( C \), then \( F(T) \subseteq F(T') \).
- (Origination) For any constraint of the form \( \langle m:T \rangle \) in \( C \), with \( V \) being a variable appearing in \( T \), there exists a preceding constraint \( \langle m':T' \rangle \) in \( C_\Lambda \), where \( V \) is a subterm of \( m' \) and \( F(T') \subseteq F(T) \).

**Overview of MSReduce**. MSReduce and MSReduce differ only in the properties of the constraint sequences that are passed to them as input.

Let \( C \) be a constraint sequence generated by algorithm 1. \( C \) can always be partitioned into two disjoint sequences of constraints \( C_A \) and \( C_P \), that we call respectively “attacker constraints” and “policy constraints”.

Formally, an attacker constraint \( \langle m:T \rangle \) models the evaluation of a query at the communication level, where \( m \) is a term that an attacker should construct from a finite set of terms \( T \). Here, both \( m \) and the terms in \( T \) are message terms, i.e. \( m \in \mathcal{T}_\Sigma(V) \) and \( T \subseteq \mathcal{T}_\Sigma(V) \).

A policy constraint models the evaluation of a query at the policy level. Differently from an attacker constraint, for a policy constraint \( \langle q:K \rangle \) the term \( q \) and the terms in \( K \) are the result of the application of function \( \zeta \) to Infons. In particular, all terms in policy constraints contain constructors belonging to \( \Sigma_{\text{Infons}} \) (cf. sections 4 and 5.1), while terms in attacker constraints never contain infon constructors. It is hence possible to distinguish between them with a simple syntactical check.

We note that \( C_A \) satisfies the monotonicity and origination properties. This is immediate from the syntax and semantics of our language; see section 3. The origination and monotonicity properties do not hold for \( C_P \), in general. However, for any constraint of the form \( \langle q:K \rangle \) in \( C_P \), with \( V \) being a variable appearing in \( K \), there exists a preceding constraint \( \langle m':T' \rangle \) in \( C_\Lambda \), where \( V \) is a subterm of \( m' \). This is because all variables in specifications of service-oriented architectures are originally instantiated at a receive event in an honest process.

We remark that the monotonicity and the origination properties hold for simple constraint sequences obtained by applying MSReduce to constraint sequences generated by algorithm 1. The resulting simple constraint sequences are thus immediately solvable. This is because:

- policy constraints, for which the monotonicity property does not hold, are eliminated by the reduction procedure. This is due to the fact that the variables in policy constraints are always subterms of applications of infon constructors. Infon constructors are uninterpreted functions for the attacker’s inference rules; hence they are never constructed or deconstructed by the reduction system. If such constraints are satisfiable, then they will be eliminated using the unification rule \( \text{un} \).

- the reduction procedure preserves the monotonicity of the attacker constraints, as shown in [15].

**B.2 Proof of termination**

The proof of termination of MSReduce follows closely the termination proof of MSReduce. The proof is based on a termination measure \((N_\text{e}, N_\text{s})\) of a constraint sequence \( C \). Here, \( N_\text{e} \) and \( N_\text{s} \) are naturals. In particular, \( N_\text{e} \) is the number of distinct variables occurring in \( C \) and \( N_\text{s} \) is a special expansion measure. Tuples are ordered lexicographically.
The expansion measure hinges upon another measure, the size $|m|$ of a term $m$, that is the number of operator applications plus the number of constants and variable in $m$. Then the expansion measure $N_e$ of constraint sequence $C$ is the sum of the expansion measures of its constraints; in turn, the expansion measure of a constraint $\langle m : T \rangle$ is $|m| \cdot \chi(T)$, where $\chi$ is defined as follows:

$$
\chi(t) = \begin{cases} 
2 & \text{if } t \text{ is a variable or constant} \\
\chi(t_1) \cdots \chi(t_n) & \text{otherwise}
\end{cases}
$$

We show now that the termination measure decreases strictly with each application of a reduction rule. Rule $\text{elim}$ removes a stand-alone variable, hence reduces $N_e$: rule $\text{un}$ either substitutes a variable, hence decreasing $N_e$, or decreases $N_e$ by removing the constraint; rules $\text{sig}$, $\text{pair}$, $\text{hash}$, $\text{penc}$ and $\text{senc}$ reduces $N_e$ by splitting the constraint in constraints whose sum of expansion measures is smaller; rules $\text{split}$ and $\text{pdec}$ decrease $N_e$ by replacing a term with terms whose product of expansion measures is smaller; rule $\text{ksub}$ substitutes a variable, hence decreases $N_e$. Finally rule $\text{sdec}$ replaces a constraint $c = \langle m : T \cup \{t\}_k \rangle$ with constraints $c' = \langle k : T \cup \{t\}_k \rangle$ and $c'' = \langle m : T \cup \{t\}_k \rangle$. The expansion measure of $c$ is $|m|\chi(T)(\chi(t)(k) + |k| + 1)$, while the product of the expansion measures of $c'$ and $c''$ is $|k|\chi(T) + |m|\chi(T)\chi(t)(k)$. Since $|k| < |m|(k) + 1$, the expansion is strictly decreased in each $\text{sdec}$ reduction step. MSReduce thus terminates.

**B.3 Proof of correctness**

In the following, for ease of presentation, we ignore the encryption hiding operator and related attacker capabilities. The encryption hiding operator is used in [15] merely to avoid non-termination, as mentioned above.

**B.3.1 Proof of soundness**

We show here that if a rule of MSReduce reduces constraint sequence $C$ to constraint sequence $C'$, then any solution $\sigma$ of $C'$ is also a solution of $C$. In other words, MSReduce does not introduce new solutions.

We condition on the rule applied:

- Rule $\text{elim}$ removes a stand-alone variable $V$ from $T \cup \{V\}$, for the active constraint being $c = \langle m : T \cup \{V\} \rangle$. We need to distinguish two cases:
  - $c$ is an attacker constraint. We show that $\mathcal{F}(T \cup V) = \mathcal{F}(T)$. It is obvious that $\mathcal{F}(T) \subseteq \mathcal{F}(T \cup \{V\})$, we show then that $\mathcal{F}(T \cup \{V\}) \subseteq \mathcal{F}(T)$. By originality property, there exists an earlier constraint $c' = \langle m' : T' \rangle$, such that $V$ does not appear in $T'$ and $V$ appears in $m'$. In particular, $m' = V$ (all constraints earlier than $c$ are simple). Due to idempotency of closure, it suffices to show that $T \cup \{V\} \subseteq \mathcal{F}(T)$, and we only need to show that $V \subseteq \mathcal{F}(T)$, since $T \subseteq \mathcal{F}(T)$. By the monotonicity property $T' \subseteq T \cup \{V\}$, and since $V \notin T'$ then $T' \subseteq T$. But $V \in \mathcal{F}(T')$, therefore also $V \in \mathcal{F}(T)$.
  - $c$ is a policy constraint. Then rule $\text{elim}$ is never applied, because all variables in $T$ appear under the application of an infon constructor. Furthermore, infon constructors are uninterpreted functions, hence cannot be deconstructed to yield the variables they contain.

- Rules $\text{split}$ and $\text{pdec}$ are sound since $\mathcal{F}$ is closed under $\partial_{\text{pair}}$ and $\partial_{\text{penc}}$.

- Rule $\text{un}$ removes $c = \langle m : T \rangle$ when $m$ is unifiable with some term $t \in T$. Let $\tau = m\nu(m,t)$. If $\sigma$ is a solution for $C'$, then $\sigma \tau$ is a solution for $C$ provided that $T\sigma \vdash m\tau$. This is obvious since $T\tau \vdash m\tau$.

- Rules $\text{pair}$, $\text{hash}$, $\text{penc}$, $\text{senc}$ and $\text{sig}$ are sound since $\mathcal{F}$ is closed under the corresponding $\partial$ rules of the attacker.

- Rule $\text{sdec}$ replaces the active constraint $c = \langle m : T \rangle$, with $\{t\}_k \in T$, with the constraints $\langle k : T \cup \{t\}_k \rangle$ and $\langle m : T \cup \{k\}_t \rangle$. This rule is sound because if $k \in \mathcal{F}(T)$, then $\mathcal{F}(T) = \mathcal{F}(T \cup \{k\})$; moreover, since $\mathcal{F}$ is closed under $\partial_{\text{sdce}}$, we have $\mathcal{F}(T) = \mathcal{F}(T \cup \{k\}_t)$ given $k \in \mathcal{F}(T)$.

**B.3.2 Proof of completeness**

We show here that for every constraint sequence $C$ and solution $\sigma$ of $C$, there exists a rule $\tau$ that reduces $C$ to $C'$, and $\sigma$ is a solution of $C'$. In other words, all solutions for $C$ are preserved in at least one reduction path.

Let $\langle m : T \rangle$ be the active constraint in $C$. The proof relies on the existence of a normal proof of $T\sigma \vdash m\sigma$, that is, a proof tree such that no label appears more than once in any path from the root to a leaf [15]:

**PROPOSITION 1.** Let $t$ be a ground term and $T$ a set of ground terms. If $t \notin T$ and $t \in \mathcal{F}(T)$ then there exists a normal sequence $\phi_1, \ldots, \phi_n$ such that $t \in \phi_n(\ldots \phi_1(T))$ and one of the following conditions holds:

- $\phi_n$ is a synthesis rule
- $\phi_1$ is an analysis rule
- $\phi_i$, for some $1 \leq i \leq n$, is $\partial_{\text{sdce}}$ and $\phi_1, \ldots, \phi_{i-1}$ are synthesis rules

The proposition intuitively states that any normal sequence $\phi_1, \ldots, \phi_n$ such that $t \in \phi_n(\ldots \phi_1(T))$ can be reordered so that analysis rules always appear earlier than synthesis rules, except in the case where $\partial_{\text{sdce}}$ is used (i.e. synthesis rules appear before $\partial_{\text{sdce}}$ to construct a non-atomic key).

**Finding an applicable rule.** Let $c = \langle m : T \rangle$ be the active constraint in $C$ and $\sigma$ a solution of $C$. Then $m\sigma \in \mathcal{F}(T\sigma)$. Intuitively, MSReduce tries to apply a sequence of reduction rules that reflects the order of a normal proof of $T\sigma \vdash m\sigma$ as indicated by proposition 1.

Recall that by definition $c$ does not contain stand-alone variables neither on the left hand side (it is chosen as the first constraint whose left hand side is not a variable) nor on the
right hand side (as all stand-alone variables are removed by application of the \textit{elim} rule). It follows that every term in \( \{m\} \cup T \) has a well-defined \textit{top level structure}, i.e. outermost function application, against which applicability of a rule can be checked.

If \( m\sigma \in T\sigma \), then rule \textit{un} is applicable. If \( m\sigma \notin T\sigma \) then we can assume a normal sequence of operators \( \phi_1, \ldots, \phi_n \) as described in proposition 1. If \( \phi_1 \) is an analysis rule then there is a term \( t \in T \) with corresponding top level structure, hence the corresponding analysis rule can be applied. Similarly for \( \phi_n \) being a synthesis rule, since \( m \) has a well-defined top structure the corresponding synthesis rule can be applied.

Finally, if \( \phi_i \) is \textit{sddec} then there is a term \( \|x\|_y \in T \) that enables rule \textit{sddec}.

\textbf{Preserving the solution.} Proving that each applicable rule preserves the solution proceeds by cases, on \( \phi_1 \) if the rule is an analysis rule, or on \( \phi_n \) if the rule is a synthesis rule. For brevity, we omit the details and explain only the proof for the case of \textit{sddec}, i.e. in which \( \phi_i \) is \textit{sddec} and \( \phi_1 \cdots \phi_{i-1} \) are all synthesis rules. Rule \textit{sddec} replaces the active constraint \( c = \langle m:T \rangle \) with constraints \( c' = \langle k:T \rangle \) and \( c'' = \langle m:T \cup \{k,t\} \rangle \), for term \( \|x\|_y \in T \). Observe that \( k\sigma \in \phi_{i-1}(\cdots \phi_1(T\sigma)) \), otherwise rule \textit{sddec} would not be applicable, and consequently \( \sigma \) is also a solution for \( c' \). Also, since \( k\sigma \in F(T\sigma) \) and \( F \) is idempotent, then \( F(T\sigma) = F(T\sigma \cup \{k\sigma, t\sigma\}) \), which shows that \( \sigma \) is a solution for \( c'' \).